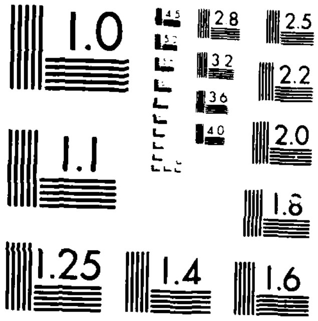
AD-A164 820 VERIFYING TEMPORAL PROPERTIES WITHOUT USING TEMPORAL LOGIC(U) CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE B ALPERN ET AL. DEC 85 CU-CSD-TR-85-723 1/1

UNCLASSIFIED N00014-86-K-0092 F/G 9/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

(12)

AD-A164 820

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION
Unclassified

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)
Cornell University TR 85-723

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Cornell University

6b OFFICE SYMBOL *(If applicable)*

7a NAME OF MONITORING ORGANIZATION
Office of Naval Research

6c ADDRESS (City, State, and ZIP Code)
Dept. of Computer Science
Cornell University
Ithaca, NY 14853

7b ADDRESS (City, State, and ZIP Code)
800 North Quincy Street
Arlington, VA 22217-5000

8a NAME OF FUNDING / SPONSORING ORGANIZATION
Office of Naval Research

8b OFFICE SYMBOL *(If applicable)*

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-86-K-0092

8c ADDRESS (City, State, and ZIP Code)
800 North Quincy Street
Arlington, VA 22217-5000

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE *(Include Security Classification)*
Verifying Temporal Properties without using Temporal Logic

12 PERSONAL AUTHOR(S)
Bowen Alpern, Fred B. Schneider

13a TYPE OF REPORT

13b TIME COVERED
FROM ________ TO ________

14 DATE OF REPORT (Year, Month, Day)
December 1985

15 PAGE COUNT
41

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
concurrent programs, temporal logic, program verification, property recognizers, Buchi automata

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

A new approach for proving temporal properties of concurrent programs is presented. The approach does not use temporal logic. To show that a program satisfies a given temporal property, the property is first decomposed into proof obligations. These obligations are then discharged by devising suitable invariant assertions and variant functions for the program. The approach is quite general - it handles a superset of the properties that can be expressed in linear-time temporal logic.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a NAME OF RESPONSIBLE INDIVIDUAL
Fred B. Schneider

22b TELEPHONE (Include Area Code)
607-255-9221

22c OFFICE SYMBOL

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

86 2 27 011

Verifying Temporal Properties without using Temporal Logic*

Bowen Alpern
Fred B. Schneider

TR 85-723
December 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work is supported in part by NSF Grant DCR-8320274 and a grant from the Office of Naval Research

Verifying Temporal Properties without using Temporal Logic[*]

Bowen Alpern
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

December 23, 1985

ABSTRACT

A new approach for proving temporal properties of concurrent programs is presented. The approach does not use temporal logic. To show that a program satisfies a given temporal property, the property is first decomposed into proof obligations. These obligations are then discharged by devising suitable invariant assertions and variant functions for the program. The approach is quite general—it handles a superset of the properties that can be expressed in linear-time temporal logic.

[*] This work is supported in part by NSF Grant DCR-8320274 and a grant from the Office of Naval Research.

1. Introduction

Experience has shown that while it may be possible to understand a sequential program by considering some subset of its executions, this is impossible for concurrent programs. Consequently, over the past 15 years, there has been increasing interest in ways to deduce properties of program behavior from the program text itself. The program text obviously contains all the information needed to decide what executions are possible. Moreover, while the number of possible executions is likely to be intractably large, only a single program text need be analyized.

An execution of a program can be viewed as a potentially infinite sequence of states called a *history*. In a history, the first state is an initial state of the program and each following state results from executing a single atomic action in the preceding state. In a concurrent or distributed program, a history is the sequence of states that results from interleaving the atomic actions of the processes as they execute.

A *property* defines a set of sequences of states; a program *satisfies* a property if each of its histories is in the set defined by the property. A property can be specified as a predicate on sequences. This allows the essence of the property to be made explicit.

Some examples of properties frequently arising in practice follow.

- *Partial Correctness* includes all sequences of program states such that, if the first state in the sequence satisfies some given precondition and the sequence is finite, then in the final state the program counter denotes the end of the program and some given postcondition is satisfied.

- *Total Correctness*, which is stronger than Partial Correctness, includes all sequences such that if the first state in the sequence satisfies some given precondition, then the sequence is finite and the value of the program counter in the final state denotes the end of the program as well as satisfying some given postcondition.

- *Mutual Exclusion* includes all sequences in which there is no state where the program counters for two or more processes denote control points inside critical sections.

- *Deadlock Freedom* includes all sequences in which there is no state where both (i) some process has no enabled atomic actions and (ii) no subsequent execution by any other process can alter that.

- *First-come First-served* includes all sequences in which processes that request service in one order are not serviced in another order.

- *Starvation Freedom* includes all sequences in which a process with an atomic action that is enabled frequently enough will make progress eventually.

Formulas of temporal logic can be interpreted as predicates on sequences of states, and various formulations of temporal logic have been used for specifying properties of interest to designers of concurrent programs [Lamport 83a] [Lamport 83b] [Manna & Pnueli 81a] [Wolper 83]. While there is not general agreement on the details of such a specification language, there is agreement that temporal logic provides a good basis for such a language and it, or something close to it, is sufficiently expressive.

Temporal logic has also been used in proving properties of concurrent programs [Pnueli 77] [Manna & Pnueli 81b] [Manna & Pnueli 84] [Owicki & Lamport 82]. Here, a program is regarded as defining a collection of temporal logic axioms. The programmer proves a property of interest by using these axioms along with program-independent axioms and inference rules of temporal logic [Manna & Pnueli 83]. Various packagings of the approach avoid the necessity of making temporal inferences by restricting the class of properties that can be proved. Examples include Hoare's logic for Partial Correctness of sequential programs [Hoare 69] and its extension to concurrent programs [Owicki & Gries 76], GHL (Generalized Hoare Logic) for proving safety properties of concurrent programs [Lamport 80] [Lamport & Schneider 84], and proof lattices for proving liveness properties [Owicki & Lamport 82].

This paper introduces a new approach for proving properties of (concurrent) programs. The approach can handle a broad class of properties, including any property that can be expressed in temporal logic. Using our approach, to prove that a program satisfies some given property, *invariance obligations* and *variance obligations* are constructed. Invariance obligations are discharged by finding certain *invariant assertions* and showing that they are preserved by execution; variance obligations are discharged by finding *variant functions* and showing that they decrease following certain events. Hoare's partial correctness logic is used to show that the invariant assertions are preserved by execution and that the variant functions are decreased by execution.

2. Specifying Properties

Our approach is based on specifying properties by using property recognizers, which are similar to Buchi automata [Eilenberg 74]. We are not advocating property recognizers as the basis for a specification language, but we have found them to be a convenient starting point for our verification method. Mechanical procedures exist to translate any temporal logic formula into a corresponding property recognizer [Alpern 86] [Wolper 84], so starting with property recognizers does not constitute a restriction. In fact, property recognizers are more expressive than most temporal logic-based specification languages—there exist properties that can be specified using property recognizers but cannot be specified in (most) temporal logics [Wolper 83].

A property recognizer accepts those sequences of program states that are in the property it specifies. Properties can contain infinite sequences as well as finite ones, so a property recognizer must be able to accept both kinds of sequences. Recall that a finite state-automaton accepts a finite sequence if and only if it halts in an accepting state after reading the final symbol [Hopcroft & Ullman 79]. A *Buchi automaton* is a finite-state automaton with an acceptance criterion that allows it to accept infinite sequences—it accepts an infinite sequence if and only if it enters an accepting state infinitely often while reading that sequence [Eilenberg 74]. A *property recognizer* is an automaton that behaves like a standard finite-state automaton for finite input sequences and like a Buchi automaton for infinite input sequences.

An example of a property recognizer, m_{compl}, is given in Figure 2.1. It defines the set of sequences consisting of a (possibly empty) prefix of states in which each state satisfies predicate $\neg P$, immediately followed by either (i) an infinite sequence of states in which P holds for each state, or (ii) a finite sequence of states in which P holds on all except the last state.

Property recognizer m_{compl} contains three *automaton states* labeled, q_0, q_1, and q_2. The *start state* is denoted by an arc with no origin, *infinite-accepting states* by concentric circles, and *finite-accepting states* by bullets (●). An infinite sequence is accepted by a property recognizer only if it causes the recognizer to be infinitely often in some infinite-accepting state. A finite sequence is accepted by the property recognizer only if it causes the recognizer to halt (at the end of its input) in some finite-accepting state. In m_{compl}, q_0 is the start state, q_1 is an infinite-accepting state, and q_2 is a finite-accepting state.

Arcs between automaton states are labeled by program state predicates called *transition predicates*. These define transitions between automaton states based on the next symbol read from the input. For example, the arc labeled P from q_0 to q_1 in m_{compl} means that whenever m_{compl} is in q_0 and the next symbol read is a program state satisfying P, then a transition to q_1 is made. If the next symbol read by a property recognizer satisfies no transition predicate on an arc emanating from the current automaton state, the input is rejected; in this case, we say the transition is *undefined* for that symbol. This is used in m_{compl} to ensure that every

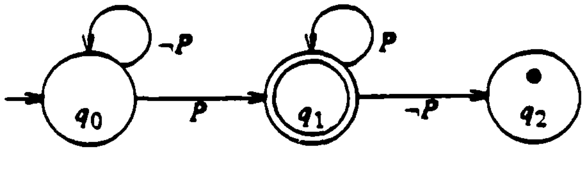


Figure 2.1. m_{compl}

finite sequence it accepts ends with a single program satisfying $\neg P$; no further transitions are possible from q_2 because there are no arcs emanating from it.

When there is more than one start state or more than one transition is possible from some automaton state for some input symbol, the property recognizer is *non-deterministic*; otherwise it is *deterministic*. Thus, m_{compl} is deterministic because it has a single start state and disjoint transition predicates label the arcs that emanate from each automaton state.

Formally, a property recognizer m for a property of a program π is a sextuple $\langle S, Q, Q_0, Q_{inf}, Q_{fin}, \delta \rangle$, where

S is the set of program states of π,
Q is the set of automaton states of m,
$Q_0 \subseteq Q$ is the set of start states of m,
$Q_{inf} \subseteq Q$ is the set of infinite-accepting states of m,
$Q_{fin} \subseteq Q$ is the set of finite-accepting states of m, and
$\delta \in (Q \times S) \rightarrow 2^Q$ is the *transition function* of m.

Transition predicates are derived from δ as follows. T_{ij}, the transition predicate associated with the arc from automaton state q_i to q_j, is the predicate that holds for all program states s such that $q_j \in \delta(q_i, s)$. Thus, T_{ij} is *false* if no symbol can cause a transition from q_i to q_j.

In order to formalize when m accepts a sequence, some definitions are required. For any sequence $\sigma = s_0 s_1 \ldots$,

$$\begin{aligned} \sigma[i] &= s_i \\ \sigma[..i] &= s_0 s_1 \ldots s_i \\ \sigma[i..] &= s_i s_{i+1} \ldots \\ |\sigma| &= \text{the length of } \sigma \ (\omega \text{ if } \sigma \text{ is infinite}). \end{aligned}$$

Transition function δ can be extended to handle finite sequences of program states:

$$\delta^*(q, \sigma) = \begin{cases} \{q\} & \text{if } |\sigma|=0 \\ \{q' \mid q^* \in \delta(q, \sigma[0]) \wedge q' \in \delta^*(q^*, \sigma[1..])\} & \text{if } 0<|\sigma|<\omega \end{cases}$$

A *run* of m for an input σ is a sequence of automaton states that m could be in while reading σ. Thus, for ρ to be a run for σ, $\rho[0] \in Q_0$, and $(\forall i: 0<i<|\sigma|: \rho[i] \in \delta(\rho[i-1], \sigma[i-1]))$. Let $\Gamma_m(\sigma)$ be the set of runs of m on σ. (It is a set because m might be non-deterministic.)

A finite sequence σ is accepted by m if and only if $\delta^*(q_0, \sigma) \cap Q_{fin} \neq \varnothing$. For an infinite sequence σ, define $INF_m(\sigma)$ to be the set of automaton states that appear infinitely often in any element of $\Gamma_m(\sigma)$. Then, σ is accepted by m if and only if $INF_m(\sigma) \cap Q_{inf} \neq \varnothing$.

Any set of finite sequences that can be recognized by a non-deterministic finite-state automaton can be recognized by some deterministic finite-state automaton [Hopcroft & Ullman 79]. Unfortunately, Buchi automata, hence property recognizers, do not enjoy this equivalence—there are sets of infinite sequences that can be recognized by non-deterministic property recognizers but by no deterministic one [Eilenberg 74]. This will ultimately require that we use different techniques for those properties specified by non-deterministic property recognizers from those specified by deterministic ones.

Examples of Property Recognizers

A property recognizer m_{pc} for Partial Correctness is shown in Figure 2.2 and one for Total Correctness, m_{tc}, is shown in Figure 2.3. In them, *Pre* is a transition predicate that holds for states satisfying the given precondition, *Done* holds for states in which the program counter denotes the end of the program, and *Post* holds for states satisfying the given postcondition.

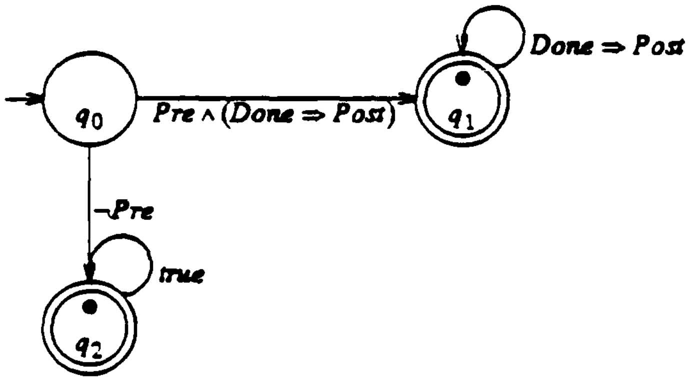


Figure 2.2. m_{pc}

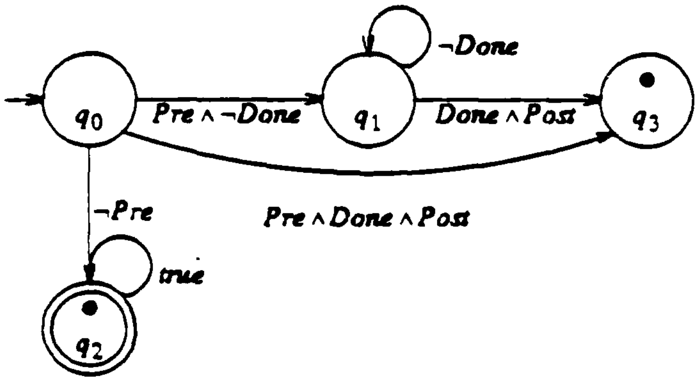


Figure 2.3. m_{tc}

A property recognizer for Mutual Exclusion of two processes, m_{mutex}, is given in Figure 2.4. There, transition predicate Cs_{ϕ} (Cs_{ψ}) holds for any state in which process ϕ (ψ) is executing in its critical section.

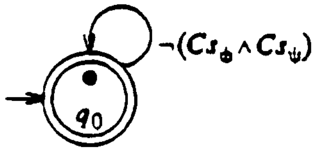


Figure 2.4. m_{mutex}

Starvation Freedom for a mutual exclusion protocol is specified by m_{starv} of Figure 2.5. A process ϕ becomes enabled when its state satisfies the predicate $Request_{\phi}$, which characterizes the state of ϕ whenever it attempts to enter its critical section, and makes progress when its state satisfies the predicate $Served_{\phi}$, which holds whenever ϕ enters its critical section. Notice that m_{starv} exploits the fact that in a mutual exclusion protocol ϕ will make but a single request for each entry into the critical section.

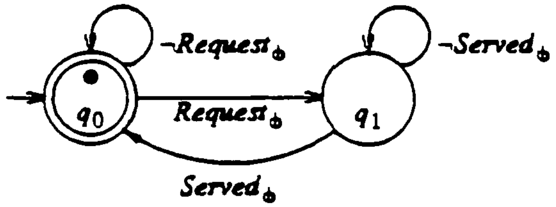


Figure 2.5. m_{starv}

3. Specifying Programs

A program π consists of a predicate $Init_{\pi}$ characterizing its initial states and a collection of atomic actions A_{π}. Presumably, $Init_{\pi}$ asserts that

- the program counter for each process in π denotes the first statement of that process, and
- other program variables have appropriate values according to any initialization in their declarations.

Knowing the atomic actions of a concurrent program is necessary in order to understand its execution, since they define the grain of interleaving of processes. The atomic actions in a process define its *control points*—the set of values that can be stored in the program counter for that process. We can denote the control points of a program by naming them within braces in the program text; this results in a *control-point annotation*. For example, program π_0 of Figure 3.1 consists of two sequential processes, ϕ and ψ, each with a single atomic action and two control points. The atomic action in process ϕ is called α_1 and the control points in ϕ are labeled 1 and 2.

Every sequential process π has a program counter pc_π. We can use this variable in describing states of the program. For example, $pc_\phi = 1 \wedge pc_\psi = 3$ defines the state of π_0 at its start and $pc_\phi = 2 \wedge pc_\psi = 4$ at its finish. The program counter of a sequential process differs from other program variables in that usually only a single process may update it and direct assignments to it are not permitted. Each atomic action, however, changes the value of the program counter. For example, atomic action α_1 in π_0 changes pc_ϕ (from 1 to 2) as well as incrementing x. The assignment to pc_ϕ by α_1, though not explicit, can be deduced from the position of α_1 in the program text.

By definition, atomic actions are executed indivisibly and to completion, so an atomic action cannot be started unless it will terminate. We therefore assume an atomic action is delayed until the state is one that will permit its termination. Using angle brackets to denote an atomic action, α_1 of π_0 is

$$\langle \textbf{if}\ pc_\phi = 1 \rightarrow pc_\phi, x := 2, x+1\ \textbf{fi} \rangle . \tag{3.1}$$

Here, we use the multiple assignment statement of [Gries 81] and the **if** of [Dijkstra 76]. The semantics of **if** require that

```
$\pi_0$: cobegin
        $\phi$: {1:}
             $\alpha_1$: $x := x+1$
             {2:}
      //
        $\psi$: {3:}
             $\alpha_2$: $x := x+1$
             {4:}
      coend
```

Figure 3.1. Simple Program

$$\mathbf{if}\ B0 \rightarrow S0 \,\square\, \ldots \,\square\, Bn \rightarrow Sn\ \mathbf{fi}$$

abort if executed in a state where none of the guards $B0$, ..., Bn holds. Thus, (3.1) is delayed until the program counter for process ϕ is 1, and then (without interruption) atomically updates the program counter and increments x. An atomic action might be delayed for reasons other than the program counter value. A P operation in process π on a general semaphore *sem*,

$$\ldots \{a{:}\}\ \mathrm{P}(sem)\ \{b{:}\} \ldots$$

defines an atomic action β:

$$\langle \mathbf{if}\ pc_\pi = a \wedge sem > 0\ \rightarrow\ pc_\pi,\ sem := b,\ sem-1\ \mathbf{fi} \rangle \tag{3.2}$$

An atomic action is *enabled* in any state where its execution would not be delayed. Let $Enabled(\alpha)$ be the set of states in which α is enabled. In Figure 3.1,

$$Enabled(\alpha_1)\ \equiv\ pc_\phi = 1$$

and in (3.2),

$$Enabled(\beta)\ \equiv\ pc_\pi = a\ \wedge\ sem > 0.$$

We can use *Enabled* to characterize states in which a program π is *blocked* and can make no further progress because there are and will be no enabled atomic actions:

$$Blocked_\pi\ \equiv\ \bigwedge_{\alpha:\alpha\in A_\pi} \neg Enabled(\alpha)$$

The effects of an atomic action α can be defined as a relation between the program state before and after it is executed. This relation can be described by a *triple* $\{P\}\ \alpha\ \{Q\}$, which is valid if executing α in a state satisfying P either does not terminate or terminates is a state satisfying Q. P is called the *precondition* and Q the *postcondition*.

Programming logics to prove validity of a triple involving a sequential program π are well known [Hoare 69]. One is summarized in Figure 3.2. If the semantics of an atomic action α is described as a sequential program, then such a logic and the following inference rule can be used to infer triples giving the semantics of α.

$$\langle\,\rangle \text{ Rule: } \quad \frac{\{P\}\ S\ \{Q\}}{\{P\}\ \langle S \rangle\ \{Q\}}$$

For example, returning to π_0 of Figure 3.1, we can establish the validity of $\{x=0\}\ \alpha_1\ \{x=1\}$ as follows:

$\{x=0\}\ pc_\phi,\ x := 2,\ x+1\ \ \{x=1\}$ (Assignment Axiom)

$\{x=0\ \wedge\ pc_\phi = 1\}\ pc_\phi,\ x := 2,\ x+1\ \ \{x=1\}$ (Rule of Consequence)

Skip Axiom: $\{P\}$ **skip** $\{P\}$

Assignment Axiom: $\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$

if Rule: $\dfrac{\{P \wedge B0\}\ S1\ \{Q\}, \quad \cdots \quad \{P \wedge Bn\}\ Sn\ \{Q\}}{\{P\}\ \ \textbf{if}\ B0 \rightarrow S0 [] \ \ldots\ [] \ Bn \rightarrow Sn\ \textbf{fi}\ \ \{Q\}}$

do Rule: $\dfrac{\{P \wedge B0\}\ S1\ \{P\}, \quad \cdots \quad \{P \wedge Bn\}\ Sn\ \{P\}}{\{P\}\ \ \textbf{do}\ B0 \rightarrow S0\ []\ \ldots\ []\ Bn \rightarrow Sn\ \textbf{od}\ \ \{P \wedge \neg B0\ \wedge\ \ldots\ \wedge\ \neg Bn\}}$

Rule of Consequence: $\dfrac{P \Rightarrow P', \quad \{P'\}\ S\ \{Q'\}, \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$

Conjunction Rule: $\dfrac{\{P\}\ S\ \{Q\}, \quad \{P'\}\ S\ \{Q'\}}{\{P \wedge P'\}\ S\ \{Q \wedge Q'\}}$

Figure 3.2. Partial Correctness Logic

$\{x=0\}$ **if** $pc_{\Phi}=1\ \rightarrow\ pc_{\Phi}, x := 2, x+1$ **fi** $\{x=1\}$ (**if** Rule)

$\{x=0\}$ $\langle$**if** $pc_{\Phi}=1\ \rightarrow\ pc_{\Phi}, x := 2, x+1$ **fi**$\rangle$ $\{x=1\}$ ($\langle\ \rangle$ Rule)

$\{x=0\}\ \ \alpha_1\ \ \{x=1\}$ (definition of α_1)

This type of reasoning, which we employ frequently in the sequel, is facilitated by the following derived rule of inference.

Atomic Action Rule: $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \langle \textbf{if}\ B \rightarrow S\ \textbf{fi}\ \rangle\ \{Q\}}$

4. Verification of Deterministic Properties

The basis for our approach to verifying that a program π satisfies a property P is the observation that if a property recognizer m for P accepts every history of π, then π satisfies P. In this section, we consider verification of properties that are specified by deterministic property recognizers; in section 8, we consider non-deterministic property recognizers. Soundness and completeness proofs are given in the Appendix.

Let m be a deterministic property recognizer for property P. One can think of m as simulating—in an abstract way—any program that satisfies P. Thus, to show that a program π satisfies m, we demonstrate such a correspondence between m and π. We do this by defining a correspondence invariant C_i for each automaton state q_i. A *correspondence invariant* C_i

for an automaton state q_i is a predicate such that C_i holds on a program state s if and only if there exists a history of π containing a program state s and m enters q_i upon reading s. Thus, if m is ever in automaton state q_i, the last program state it read must satisfy C_i. Constraints satisfied by correspondence invariants are defined inductively, as follows.

For the base case, initially, m is in state q_0 and π is in a state characterized by $Init_\pi$. Suppose that upon reading s_0, the first program state of some history of π, m enters automaton state q_j. Thus, s_0 satisfies $Init_\pi$ and T_{0j}, the transition predicate labeling the edge that connects q_0 and q_j. Therefore, C_j must satisfy $(Init_\pi \wedge T_{0j}) \Rightarrow C_j$; for any automaton state q_j entered upon reading the first symbol of any history of π, we require

$$(\forall j\colon\ (Init_\pi \wedge T_{0j}) \Rightarrow C_j). \qquad (4.1)$$

Next we must prove the induction step. Assume that if m enters automaton state q_i upon reading program state s_k in a history of π and $0 \leq k < K$, then s_k satisfies C_i. Consider the case when m reads s_K. Suppose m is in state q_i and that upon reading program state s_K, a transition is made to automaton state q_j. By the induction hypothesis s_{K-1} satisfies C_i and s_K satisfies transition predicate T_{ij}. The appropriate correspondence invariant C_j will hold provided $\{C_i\}\ \alpha\ \{T_{ij} \Rightarrow C_j\}$ is valid for any α, an atomic action of π. (If α is not enabled in s_{K-1} then the triple is trivially valid.) Generalizing to handle any atomic action and any automaton state that m might be in when s_K is read, we require:

$$\begin{array}{l} \text{For all } \alpha\colon\ \alpha \in A_\pi\colon \\ \quad \text{For all } i\colon\ q_i \in Q\colon \\ \quad\quad \{C_i\}\ \alpha\ \{\bigwedge_{j:q_j \in Q} (T_{ij} \Rightarrow C_j)\} \end{array} \qquad (4.2)$$

Thus, any collection of predicates satisfying (4.1) and (4.2) are correspondence invariants for m and π.

In order to establish that π satisfies P, we must show that every history of π is accepted by m. There are exactly three ways that m might fail to accept a history σ of π:

(1) m attempts an undefined transition when reading σ.

(2) If σ is finite, m halts in a non-finite-accepting state.

(3) If σ is infinite, m never enters an infinite-accepting state after some finite prefix of σ.

Thus, in order to prove that every history of π satisfies P, it suffices to show that (1)−(3) are impossible.

Two obligations ensure that (1) is impossible. First, we must show that m can make some transition from its start state upon reading the first program state in a history:

$$Init_{\pi} \Rightarrow \bigvee_{j:q_j \in Q} T_{0j} \tag{4.3}$$

Second, we must show that m can always make a transition upon reading subsequent states in a history. If m is in state q_i then the program state just read by m satisfies a correspondence invariant C_i. To avoid an undefined transition, any atomic action α that is then executed must transform the program state so that one of the transition predicates T_{ij} emanating from q_i holds. This is guaranteed by

$$\begin{array}{l} \text{For all } \alpha\text{: } \alpha \in A_{\pi}\text{:} \\ \quad \text{For all } i\text{: } q_i \in Q\text{:} \\ \quad\quad \{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q} T_{ij}\} \end{array} \tag{4.4}$$

We can exploit the fact that m is deterministic to combine and simplify the obligations derived so far. In a deterministic property recognizer, the transition predicates on arcs emanating from any automaton state are disjoint. Thus,

$$(\forall i,j,k\text{: } q_i, q_j, q_k \in Q \wedge j \neq k\text{: } (T_{ij} \wedge T_{ik}) = \mathit{false}). \tag{4.5}$$

Using (4.5), we combine (4.1) and (4.3), to obtain

$$\textbf{Simulation Basis: } Init_{\pi} \Rightarrow (\bigvee_{j:q_j \in Q} (T_{0j} \wedge C_j)), \tag{4.6}$$

and combine (4.2) and (4.4), to obtain

$$\begin{array}{ll} \textbf{Simulation Induction:} & \text{For all } \alpha\text{: } \alpha \in A_{\pi}\text{:} \\ & \quad \text{For all } i\text{: } q_i \in Q\text{:} \\ & \quad\quad \{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q} (T_{ij} \wedge C_j)\}\ . \end{array} \tag{4.7}$$

To ensure that it is impossible for m to halt in a non-finite-accepting state—(2) above—the correspondence invariant for any non-finite-accepting state must hold only for program states in which subsequent execution by π is inevitable. Since C_i holds of the last program state read by m, and $Blocked_{\pi}$ holds for all program states of π in which subsequent execution is not possible, we require

$$\textbf{Finite Acceptance: } (\forall i\text{: } q_i \in Q - Q_{fin}\text{: } C_i \Rightarrow \neg Blocked_{\pi}). \tag{4.8}$$

Finally, we ensure that (3) is impossible. A set Q' of automaton states is *strongly connected* if and only if there is a sequence of transitions from any element of Q' to any other without involving an automaton state outside of Q'. A *reject knot* κ is a maximal strongly connected subset of Q containing no infinite-accepting states. It may, however, contain finite-accepting states. In order to show that (3) is impossible, we must prove that no run for an infinite history of π is restricted to automaton states in $Q - Q_{inf}$. We do this by constructing a *variant function* v_{κ} for each reject knot κ.

A variant function $v_\kappa(q,s)=0$ is a function from automaton and program states to some well-founded set.[1] For simplicity, assume that this well-founded set is the Natural Numbers. We require that whenever $v_\kappa(q,s)=0$ for any automata state q and program state s, either q is not in κ or else q is a finite-accepting state and s is the last state in the history.

Knot Exit: $(\forall i\colon\ q_i \in \kappa\colon\ (v_\kappa(q_i)=0) \Rightarrow \mathit{Blocked}_\pi \vee \neg C_i)$ (4.9)

This means that if $v_\kappa(q)=0$, either the history is finite and will be accepted by m or an infinite-accepting state has just been entered since the property recognizer is no longer in κ. Finally, to ensure that the variant function does reach 0, we require that it is decreased by every atomic action in π that might be executed:

Knot Variance: For all α: $\alpha \in A_\pi$:

$$\begin{array}{l} \text{For all } q_i \in \kappa: \\ \{C_i \wedge 0 < v_\kappa(q_i) = V\}\ \alpha\ \{\bigwedge_{j:q_j\in\kappa} ((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V)\} \end{array} \qquad (4.10)$$

Note that requiring that $v_\kappa(q)$ be decreased by execution of any eligible atomic action does not preclude proving properties under various fairness assumptions. To prove that a property P holds assuming some fairness property F holds, a property recognizer for $F \Rightarrow P$ is constructed and proof obligations are extracted from it. Standard techniques exist to construct a property recognizer for $F \Rightarrow P$ from property recognizers for F and P [Eilenberg 74].

The five proof obligations—Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10)—are of three basic forms. Simulation Basis (4.6), Finite Acceptance (4.8), and Knot Exit (4.9) involve proving that predicate logic formulas are valid. Simulation Induction (4.7) involves proving invariance of some assertions. Knot Variance (4.10) involves proving that certain events cause variant functions to be decreased. Of course, the intellectual challenge in proving that a program satisfies a property lies not in checking the proof obligations, but in devising the correspondence invariants and variant functions. The proof obligations, however, do give insight into forms the correspondence invariants and variant function might take. In particular, the proof obligations define a collection of equations whose unknowns are the correspondence invariants and variant functions. Solving the equations—admittedly a difficult task—would provide the desired correspondence invariants and variant functions.

5. A Detailed Example

To illustrate our verification method, we prove that if program π_0 of Figure 3.1 is started in a state where $x=0$ then it will terminate with $x=2$. This is an instance of Total Correctness.

[1] The program state argument is often left implicit.

For π_0, we have

$$Init_{\pi_0} \equiv pc_{\phi}=1 \wedge pc_{\psi}=3$$

$$Blocked_{\pi_0} \equiv pc_{\phi}=2 \wedge pc_{\psi}=4$$

and $A_{\pi_0}=\{\alpha_1,\ \alpha_2\}$, where

$$\alpha_1 \equiv \langle \textbf{if}\ pc_{\phi}=1 \rightarrow pc_{\phi},\ x := 2,\ x+1\ \textbf{fi} \rangle$$

$$\alpha_2 \equiv \langle \textbf{if}\ pc_{\psi}=3 \rightarrow pc_{\psi},\ x := 4,\ x+1\ \textbf{fi} \rangle.$$

A property recognizer m_{tc} for Total Correctness appears in Figure 2.3. For predicates *Pre*, *Post*, and *Done* we choose:

$$Pre \equiv x=0$$

$$Post \equiv x=2$$

$$Done \equiv pc_{\phi}=2 \wedge pc_{\psi}=4$$

Thus, m_{tc} accepts every sequence of states such that if $x=0$ holds for the first state, then the sequence is finite and the final state is one in which $x=2$ and both ϕ and ψ have terminated.

We first define correspondence invariants for each of the four automaton states of m_{tc}.

$$C_0 \equiv false$$

$$\begin{aligned} C_1 \equiv\ & pc_{\phi}=1 \Rightarrow ((pc_{\psi}=3 \Rightarrow x=0) \wedge (pc_{\psi}=4 \Rightarrow x=1)) \ \wedge \\ & pc_{\phi}=2 \Rightarrow ((pc_{\psi}=3 \Rightarrow x=1) \wedge pc_{\psi} \neq 4)) \ \wedge \\ & pc_{\psi}=3 \Rightarrow ((pc_{\phi}=1 \Rightarrow x=0) \wedge (pc_{\phi}=2 \Rightarrow x=1)) \ \wedge \\ & pc_{\psi}=4 \Rightarrow ((pc_{\phi}=1 \Rightarrow x=1) \wedge pc_{\phi} \neq 2)) \end{aligned}$$

$$C_2 \equiv true$$

$$C_3 \equiv pc_{\phi}=2 \wedge pc_{\psi}=4 \wedge x=2$$

To satisfy Simulation Basis (4.6), we must show that

$$Init_{\pi_0} \Rightarrow ((false \wedge C_0) \vee (Pre \wedge \neg Done \wedge C_1) \vee (\neg Pre \wedge C_2) \vee (Pre \wedge Done \wedge Post \wedge C_3)$$

is valid. Substituting, we get

$$\begin{aligned} & (pc_{\phi}=1 \ \wedge \ pc_{\psi}=3) \\ & \quad \Rightarrow (false \vee (x=0 \wedge \neg (pc_{\phi}=2 \wedge pc_{\psi}=4) \wedge C_1) \vee (x \neq 0) \vee (x=0 \wedge pc_{\phi}=2 \wedge pc_{\psi}=4 \wedge x=2)), \end{aligned}$$

which is valid.

To satisfy Simulation Induction (4.7), we must show for each $\alpha \in A_{\pi_0}$ that the following triples are valid:

$$\{C_0\} \;\; \alpha \;\; \{(T_{00} \wedge C_0) \vee (T_{01} \wedge C_1) \vee (T_{02} \wedge C_2) \vee (T_{03} \wedge C_3)\} \tag{5.1}$$

$$\{C_1\} \;\; \alpha \;\; \{(T_{10} \wedge C_0) \vee (T_{11} \wedge C_1) \vee (T_{12} \wedge C_2) \vee (T_{13} \wedge C_3)\} \tag{5.2}$$

$$\{C_2\} \;\; \alpha \;\; \{(T_{20} \wedge C_0) \vee (T_{21} \wedge C_1) \vee (T_{22} \wedge C_2) \vee (T_{23} \wedge C_3)\} \tag{5.3}$$

$$\{C_3\} \;\; \alpha \;\; \{(T_{30} \wedge C_0) \vee (T_{31} \wedge C_1) \vee (T_{32} \wedge C_2) \vee (T_{33} \wedge C_3)\} \tag{5.4}$$

Since the triples for α_2 are symmetric with those for α_1, we prove only the former.

Triple (5.1) is valid because $C_0 = \mathit{false}$ and $\{\mathit{false}\}\ \alpha\ \{R\}$ is valid for any R.

Substituting for the transition predicates in (5.2) and simplifying yields

$$\{C_1\} \;\; \alpha_1 \;\; \{(\neg \mathit{Done} \wedge C_1) \vee (\mathit{Done} \wedge \mathit{Post} \wedge C_3)\}. \tag{5.5}$$

From definition (3.1) of α_1 and the Atomic Action Rule, to prove the validity of (5.5), it suffices to demonstrate the validity of

$$\{C_1 \wedge pc_\Phi = 1\} \;\; pc_\Phi,\, x := 2,\, x+1 \;\; \{(\neg \mathit{Done} \wedge C_1) \vee (\mathit{Done} \wedge \mathit{Post} \wedge C_3)\}.$$

Expanding and substituting, this is

$$\begin{aligned}
&\{(pc_\Psi = 3 \Rightarrow x = 0) \wedge (pc_\Psi = 4 \Rightarrow x = 1) \wedge pc_\Phi = 1\} \\
&pc_\Phi,\, x := 2,\, x+1 \\
&\{(\neg (pc_\Phi = 2 \wedge pc_\Psi = 4) \wedge C_1) \vee (pc_\Phi = 2 \wedge pc_\Psi = 4 \wedge x = 2)\}
\end{aligned}$$

and follows from the Assignment Axiom and Rule of Consequence.

Triple (5.3) simplifies to $\{\mathit{true}\}\ \alpha_1\ \{\mathit{true}\}$ because $C_2 = T_{22} = \mathit{true}$ and is valid.

Triple (5.4) simplifies to $\{C_3\}\ \alpha_1\ \{\mathit{false}\}$ because T_{30}, T_{31}, T_{32}, and T_{33} are all *false*—those transitions are not possible in m_{tc}. From definition of α_1 (3.1) and the Atomic Action Rule, to prove (5.4) it suffices to show validity of

$$\{C_3 \wedge pc_\Phi = 1\} \;\; pc_\Phi,\, x := 2,\, x+1 \;\; \{\mathit{false}\}.$$

Since $(C_3 \wedge pc_\Phi = 1) = \mathit{false}$, this reduces to $\{\mathit{false}\}\ pc_\Phi,\, x := 2,\, x+1\ \{\mathit{false}\}$ which is valid.

To satisfy Finite Acceptance (4.8), since $Q_{fin} = \{q_2, q_3\}$ we must prove that

$$(C_0 \Rightarrow \neg \mathit{Blocked}_{\pi_\sigma}) \;\; \wedge \;\; (C_1 \Rightarrow \neg \mathit{Blocked}_{\pi_\sigma}).$$

Substituting and simplifying, we get

$$(\mathit{false} \Rightarrow (\neg \mathit{Blocked}_{\pi_\sigma})) \;\; \wedge \;\; (C_1 \Rightarrow (pc_\Phi \neq 2 \vee pc_\Psi \neq 4)),$$

which is valid.

The final two obligations concern reject knots. There is a single reject knot $\kappa = \{q_1\}$ in m_{tc}. Define

$$v_\kappa(q_1) = (2-pc_\phi)+(4-pc_\psi).$$

Knot Exit (4.9) requires that

$$(v_\kappa(q_1)=0) \Rightarrow \mathit{Blocked}_{\pi_0} \vee \neg C_1.$$

This is valid because

$$\begin{aligned}(v_\kappa(q_1)=0) &\Rightarrow (pc_\phi=2 \wedge pc_\psi=4)\\ &= \mathit{Blocked}_{\pi_0}.\end{aligned}$$

To satisfy Knot Variance (4.10), we must establish the validity of 2 triples:

$$\{C_1 \wedge 0<v_\kappa(q_1)=V\} \;\; \alpha_1 \;\; \{(\neg \mathit{Done} \wedge C_1) \Rightarrow v_k(q_1)<V\} \tag{5.6}$$

$$\{C_1 \wedge 0<v_\kappa(q_1)=V\} \;\; \alpha_2 \;\; \{(\neg \mathit{Done} \wedge C_1) \Rightarrow v_k(q_1)<V\} \tag{5.7}$$

We give details only for the first; the second is similar. Using definition (3.1) of α_1, the Atomic Action Rule, and the Rule of Consequence, to prove (5.6) it suffices to prove

$$\{C_1 \wedge 0<v_\kappa(q_1)=V \wedge pc_\phi=1\} \;\; pc_\phi,\, x := 2,\, x+1 \;\; \{v_\kappa(q_1)<V\}.$$

This is valid because changing pc_ϕ from 1 to 2 decreases v_κ.

6. Property Outlines

A *property outline* provides a compact representation of the correspondence invariants and the Simulation Induction (4.7) obligations for a given property recognizer and program. Property outlines play much the same role in our approach to verification as proof outlines do for verifying Partial Correctness using Hoare's partial correctness logic—they make it easy to do verification informally and make it easy to present a proof. In fact, proof outlines and property outlines are closely related, as we show in section 6.4.

6.1. Proof Outlines

A *proof outline* for a concurrent program π is the text of π annotated with an *assertion* P^{cp} at each control point cp. Each assertion is a first-order predicate logic formula involving the program variables and program counters of π.[2] A proof outline is *valid* provided:

Proof Outline Validity: Executing any enabled atomic action in a state where the assertions associated with the control points denoted by program counters hold produces a state in which the assertions associated with the control points denoted by program counters still hold.

Proving validity of a proof outline for a concurrent program can be reduced to proving

[2] The conjunct $pc_\phi=cp$ is often left implicit and omitted from P^{cp} in a proof outline for process ϕ.

the validity of a collection of triples [Owicki & Gries 76].[3] This is done as follows, where $pre(\alpha)$ is the assertion immediately preceding α in the proof outline and $post(\alpha)$ is the assertion immediately following it.

Sequential Correctness: For each atomic action α in the proof outline, prove

$$\{pre(\alpha)\}\ \alpha\ \{post(\alpha)\}.$$

Interference Freedom: For each atomic action α in the proof cutline and every assertion R in a process different from the one containing α, prove:

$$\{pre(\alpha) \wedge R\}\ \alpha\ \{R\}.$$

6.2. Property Outlines

A property outline for property recognizer m and program π is obtained by adding information about correspondence invariants to a control-point annotation for π. For each control point cp, we specify for every automaton state q of m what must hold when the program counter denotes cp if the property recognizer is in a state q. This is done by placing a property assertion at each control point in a control-point annotation for π.

A *property assertion* has the form

$\tilde{P}$: $q_0 \sim P_0 \mid q_1 \sim P_1 \mid \ldots \mid q_n \sim P_n$,

where $\tilde{P}$ is a label, $q_0, q_1, \ldots, q_n$ are the automaton states of m, and $P_0, P_1, \ldots, P_n$ are first-order predicate logic formulas involving the program variables of π (possibly including program counters). $\tilde{P}$ *holds* in an automaton state q_i and program state s if s satisfies P_i. A property outline for π and m is *valid* provided:

Property Outline Validity: Executing any enabled atomic action in an automaton state q and program state s where the property assertions associated with the control points denoted by program counters hold produces a program state s' that causes the property recognizer to make a transition to an automaton state q' in which the property assertions associated with the control points denoted by program counters still hold.

Figure 6.1 is a valid property outline for m_{tc} (Total Correctness) and π_0 (of Figure 3.1).

We can exploit the similarity in the definition of validity for proof outlines and for property outlines in developing a procedure to prove validity of a property outline. Define a *property triple*

$$\{\tilde{P}:\ q_0 \sim P_0 \mid \ldots \mid q_n \sim P_n\}\ \alpha\ \{\tilde{Q}:\ q_0 \sim Q_0 \mid \ldots \mid q_n \sim Q_n\}, \qquad (6.1)$$

[3] If an atomic action like "!" or "?" of CSP spans more than one process, then a third obligation, variously called *satisfaction* or *cooperation* must also be satisfied. Our results for property outlines can also be generalized along these lines.

```
$\pi_0$: cobegin
    $\phi$: {1: $q_0 \sim \mathit{false} \mid q_1 \sim (pc_\psi = 3 \Rightarrow x = 0) \wedge (pc_\psi = 4 \Rightarrow x = 1) \mid$
            $q_2 \sim \mathit{true} \mid q_3 \sim \mathit{false}$}
        $\alpha_1$: $x := x+1$
        {2: $q_0 \sim \mathit{false} \mid q_1 \sim (pc_\psi = 3 \Rightarrow x = 1) \wedge pc_\psi \neq 4 \mid$
            $q_2 \sim \mathit{true} \mid q_3 \sim pc_\phi = 2 \wedge pc_\psi = 4 \wedge x = 2$}
  //
    $\psi$: {3: $q_0 \sim \mathit{false} \mid q_1 \sim (pc_\phi = 1 \Rightarrow x = 0) \wedge (pc_\phi = 2 \Rightarrow x = 1) \mid$
            $q_2 \sim \mathit{true} \mid q_3 \sim \mathit{false}$}
        $\alpha_2$: $x := x+1$
        {4: $q_0 \sim \mathit{false} \mid q_1 \sim (pc_\phi = 1 \Rightarrow x = 1) \wedge pc_\phi \neq 2 \mid$
            $q_2 \sim \mathit{true} \mid q_3 \sim pc_\phi = 2 \wedge pc_\psi = 4 \wedge x = 2$}
  coend
```

Figure 6.1. Example Property Outline

to be *valid* if execution of α in an automaton state q_i and program state satisfying P_i either does not terminate or terminates in a program state s such that (i) s causes the property recognizer to make a transition to automaton state q_j and (ii) s satisfies Q_j. Note that (6.1) cannot be a partial correctness logic triple because it contains property assertions in its pre- and postcondition. However, the interpretation of (6.1) is quite similar to the interpretation of a partial correctness logic triple. In fact, if we can show how to establish the validity of a property triple like (6.1) and one like

$$\{\tilde{P} \wedge \tilde{R}\} \;\; \alpha \;\; \{\tilde{R}\}, \tag{6.2}$$

where $\tilde{P}$ and $\tilde{R}$ are property assertions, then we have solved the problem of establishing the validity of a property outline. This is because we can then use Sequential Correctness and Interference Freedom to reduce the problem to showing that a collection of property triples are valid. The soundness of this approach for establishing property outline validity is based on the same argument as for proof outline validity.

Based on the interpretation of property assertions, note that:

$$\begin{aligned}((q_0 \sim P_0 \mid \ldots \mid q_n \sim P_n) \;\wedge\; (q_0 \sim R_0 \mid \ldots \mid q_n \sim R_n)) \\ = \;\; (q_0 \sim P_0 \wedge R_0 \mid \ldots \mid q_n \sim P_n \wedge R_n)\end{aligned} \tag{6.3}$$

Thus, it suffices to be able to prove the validity of property triples like (6.1) since using (6.3), those like (6.2) can always be transformed to be like (6.1). We therefore turn to the problem of proving validity of property triples.

To prove the validity of (6.1), it suffices to prove the following partial correctness logic triples.

$$\{P_0\} \;\; \alpha \;\; \{(T_{00} \wedge Q_0) \vee \ldots \vee (T_{0n} \wedge Q_n)\} \tag{6.4}$$

$$\{P_1\} \;\; \alpha \;\; \{(T_{10} \wedge Q_0) \vee \ldots \vee (T_{1n} \wedge Q_n)\} \tag{6.5}$$

$$\ldots$$

$$\{P_n\} \;\; \alpha \;\; \{(T_{n0} \wedge Q_0) \vee \ldots \vee (T_{nn} \wedge Q_n)\} \tag{6.6}$$

The first, (6.4), establishes that execution of α in a state satisfying P_0 either does not terminate or terminates in a state satisfying $T_{0j} \wedge Q_j$, for some j. From this, we conclude that execution of α in a state satisfying P_0 with m in automaton state q_0 either does not terminate or terminates in a state s' satisfying $T_{0j} \wedge Q_j$ and m makes a transition to automaton state q_j upon reading this (next) symbol in the history being generated by π. Thus, $\tilde{Q}$ holds for the case that m is started in q_0. Repeating this argument for the remaining triples, we find that no matter what automaton state m is in when α is executed, $\tilde{Q}$ will hold if α terminates. Thus, (6.4)−(6.6) together imply that executing α in a state satisfying $\tilde{P}$ either does not terminate or terminates in a state satisfying $\tilde{Q}$, hence $\{\tilde{P}\}\ \alpha\ \{\tilde{Q}\}$.

We illustrate this approach for proving validity of a property outline, on the one in Figure 6.1. There are two Sequential Correctness obligations:

$$\{1\}\ \alpha_1\ \{2\} \tag{6.7}$$

$$\{3\}\ \alpha_2\ \{4\} \tag{6.8}$$

And, there are four Interference Freedom obligations:

$$\{1 \wedge 3\}\ \alpha_1\ \{3\} \tag{6.9}$$

$$\{1 \wedge 4\}\ \alpha_1\ \{4\} \tag{6.10}$$

$$\{3 \wedge 1\}\ \alpha_2\ \{1\} \tag{6.11}$$

$$\{3 \wedge 2\}\ \alpha_2\ \{2\} \tag{6.12}$$

The details for only one of these property triples will be given; the remaining ones are left to the energetic reader. Property triple (6.7) is:

$\{1\colon\ q_0\sim \mathit{false}\ |\ q_1\sim (pc_\psi=3 \Rightarrow x=0)\wedge(pc_\psi=4 \Rightarrow x=1)\ |$
$\quad q_2\sim \mathit{true}\ |\ q_3\sim \mathit{false}\}$
$\alpha_1\colon\ x := x+1$
$\{2\colon\ q_0\sim \mathit{false}\ |\ q_1\sim (pc_\psi=3 \Rightarrow x=1)\wedge pc_\psi\neq 4\ |$
$\quad q_2\sim \mathit{true}\ |\ q_3\sim pc_\phi=2\wedge pc_\psi=4\wedge x=2\}$

Decomposing this into partial correctness logic triples we get:

$$\begin{array}{l}\{\mathit{false}\}\\ \alpha_1\\ \{(\mathit{Pre}\wedge\neg\,\mathit{Done}\wedge(pc_\psi=3\Rightarrow x=1)\wedge pc_\psi\neq 4)\vee(\neg\,\mathit{Pre})\\ \ \vee(\mathit{Pre}\wedge\mathit{Done}\wedge\mathit{Post}\wedge pc_\phi=2\wedge pc_\psi=4\wedge x=2)\}\end{array} \tag{6.13}$$

$$\begin{array}{l}\{(pc_\psi=3\Rightarrow x=0)\wedge(pc_\psi=4\Rightarrow x=1)\}\\ \alpha_1\\ \{(\neg\,\mathit{Done}\wedge(pc_\psi=3\Rightarrow x=1)\wedge pc_\psi\neq 4)\vee(\mathit{Done}\wedge\mathit{Post}\wedge pc_\phi=2\wedge pc_\psi=4\wedge x=2)\}\end{array} \tag{6.14}$$

$$\{\mathit{true}\}\ \ \alpha_1\ \ \{\mathit{true}\} \tag{6.15}$$

$$\{\mathit{false}\}\ \ \alpha_1\ \ \{\mathit{false}\} \tag{6.16}$$

Triples (6.13) and (6.16) follow trivially because the precondition of each is *false*; (6.14) follows from the Assignment Axiom; and (6.15) follows because the postcondition is *true*.

6.3. Proof Obligations and Property Outlines

The proof obligations of section 4 are based on using correspondence invariants that link program states and property recognizer states. Therefore, to show that π satisfies m using a property outline $\hat{PO}$ for m and π, we must be able to extract from $\hat{PO}$ the correspondence invariant for each automaton state of m. Doing this turns out to be trivial, due to the way property assertions are defined. Each property assertion in a property outline contains a piece of every correspondence invariant. These pieces are labeled by the automaton state to which they correspond (by the "$q\sim$") and are exactly the part of the correspondence invariant that must hold whenever a program counter denotes the control point to which the property assertion is attached.

Given a program π consisting of a set of processes $PROC_\pi$, let CP_ϕ be the set of control points in process ϕ for $\phi \in PROC_\pi$. Suppose the property assertion attached to control point cp in a valid property outline for π and m is of the form $(q_0\sim P_0^{cp}\ |\ q_1\sim P_1^{cp}\ |\ \ldots\ |\ q_n\sim P_n^{cp})$. Then, choose

$$C_i \ \equiv \bigwedge_{\phi\in PROC_\pi}\ \bigwedge_{cp\in CP_\phi}(pc_\phi=cp\Rightarrow P_i^{cp}) \tag{6.17}$$

as the correspondence invariant for automaton state q_i. This choice eliminates the need to demonstrate Simulation Induction (4.7)—this obligation is subsumed by having established validity of the property outline, as we now show.

Consider an atomic action from a process ϕ,

$$\alpha\colon \langle \textbf{if}\ pc_{\phi}=cp\ \rightarrow\ \bar{x},\, pc_{\phi} := \bar{e},\, cp'\ \textbf{fi}\,\rangle$$

where $\bar{x}$ is a vector of the program variables changed by executing α and $\bar{e}$ is a vector of expressions whose values are assigned to those variables. Simulation Induction (4.7) requires that we prove, for each automaton state q_i,

$$\{C_i\}\ \ \alpha\ \ \{(T_{i0} \wedge C_0) \vee \ldots \vee (T_{in} \wedge C_n)\}$$

According to the Atomic Action Rule and Rule of Consequence, this is implied by

$$\{C_i \wedge pc_{\phi}=cp\}\ \ \bar{x},\, pc_{\phi} := \bar{e},\, cp'\ \ \{pc_{\phi}=cp'\ \wedge\ ((T_{i0} \wedge C_0) \vee \ldots \vee (T_{in} \wedge C_n))\}. \qquad (6.18)$$

The precondition and postcondition of (6.18) can be simplified because $(C_i \wedge pc_{\phi}=cp) = (C_i^{-\phi} \wedge pc_{\phi}=cp \wedge P_i^{cp})$, where

$$C_i^{-\phi}\ \ \equiv\ \ \bigwedge_{\substack{\psi \in PROC_{\pi} \\ \psi \neq \phi}}\ \bigwedge_{cp \in CP_{\psi}} (pc_{\psi}=cp \Rightarrow P_i^{cp}),$$

so we have

$$\begin{array}{l} \{P_i^{cp} \wedge C_i^{-\phi} \wedge pc_{\phi}=cp\} \\ \bar{x},\, pc_{\phi} := \bar{e},\, cp' \\ \{pc_{\phi}=cp'\ \wedge\ ((T_{i0} \wedge P_0^{cp} \wedge C_0^{-\phi}) \vee \ldots \vee (T_{in} \wedge P_n^{cp} \wedge C_n^{-\phi}))\}. \end{array} \qquad (6.19)$$

Therefore, due to the Conjunction Rule and the fact that transition predicates are disjoint, it suffices to prove

$$\{P_i^{cp} \wedge pc_{\phi}=cp\}\ \ \bar{x},\, pc_{\phi} := \bar{e},\, cp'\ \ \{pc_{\phi}=cp'\ \wedge\ ((T_{i0} \wedge P_0^{cp'}) \vee \ldots \vee (T_{in} \wedge P_n^{cp'}))\}, \text{ and} \qquad (6.20)$$

$$\{P_i^{cp} \wedge C_i^{-\phi} \wedge pc_{\phi}=cp\}\ \ \bar{x},\, pc_{\phi} := \bar{e},\, cp'\ \ \{pc_{\phi}=cp' \wedge ((T_{i0} \wedge C_0^{-\phi}) \vee \ldots \vee (T_{in} \wedge C_n^{-\phi}))\} \qquad (6.21)$$

Notice, (6.20) is exactly what was proved in the Sequential Correctness step of establishing validity of $P\bar{O}$. Now we prove (6.21). Using the Conjunction Rule and the definition of $C_i^{-\phi}$, it suffices to prove:

For all ψ: $\psi \neq \phi\ \wedge\ \psi \in PROC_{\pi}$

  For all c: $c \in CP_{\psi}$:

$$\{P_i^{cp} \wedge P_i^{c} \wedge pc_{\phi}=cp\}\ \ \bar{x},\, pc_{\phi} := \bar{e},\, cp'\ \ \{pc_{\phi}=cp'\ \wedge\ ((T_{i0} \wedge P_0^{c}) \vee \ldots \vee (T_{in} \wedge P_n^{c}))\}$$

And, these triples are exactly what was proved in the Interference Freedom step of establishing validity of $P\bar{O}$.

Thus, given a valid property outline for m and π, in order to prove that π satisfies m, extract the correspondence invariants from the property outline and prove Simulation Basis (4.6), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10)—Simulation Induction (4.7) follows immediately from validity of the property outline.

6.4. Proof Outlines Revisited

Proof outlines for partial correctness logic can be formulated as property outlines. Let PO_{pcl} be a valid proof outline for a concurrent program π where assertion P^{cp} is associated with each control point cp in π. A valid property outline $P\mathring{O}_{prop}$ that embodies the information in PO_{pcl} is one in which each control point cp has associated with it a property assertion $q_0 \sim P^{pc}$. $P\mathring{O}_{prop}$ is for m_{triv} (given in Figure 6.2) and π.

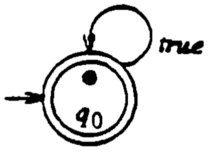


Figure 6.2. m_{triv}

Validity of $P\mathring{O}_{prop}$ follows from the partial correctness logic triples for Sequential Correctness and Interference Freedom used to establish validity of PO_{pcl}.

7. Mutual Exclusion Example

Solving the mutual exclusion problem involves devising protocols to ensure that two or more processes do not execute in *critical sections* at the same time. A good solution to the mutual exclusion protocol must not only satisfy this Mutual Exclusion property, but should ensure that a process attempting to enter a critical section eventually does so, assuming no process remains forever in its critical section—Starvation Freedom. We might also require that a protocol satisfy First-come First-served, which asserts that requests to enter a critical section are not served out-of-order.

In this section, we prove that a program *crits* based on the two-process mutual exclusion protocol in [Peterson 81] satisfies Mutual Exclusion, Starvation Freedom, and First-come First-served. The interested reader might wish to compare our proofs with the operational proofs for Mutual Exclusion and Starvation Freedom in [Peterson 81] and the temporal logic proofs for those properties in [Pnueli 86] and for First-come First-served in [Pnueli & Manna 83].

A control-point annotation for the program is given in Figure 7.1. Assume that initially $active_\phi = active_\psi = false$, since neither ϕ nor ψ is initially executing in its critical section, and that $turn$ is initialized to ϕ or ψ. Thus,

$$Init_{crits} \equiv pc_\phi=1 \wedge pc_\psi=8 \wedge \neg active_\phi \wedge \neg active_\psi \wedge (turn=\phi \vee turn=\psi)$$

$$Blocked_{crits} \equiv active_\phi \wedge active_\psi \wedge turn \neq \phi \wedge turn \neq \psi.$$

```
crits: cobegin
    $\phi$: {1:} do true $\rightarrow$ {2:}
                       non critical section;
                       {3:}
                       $active_{\phi}$ := true;
                       {4:}
                       turn := $\psi$;
                       {5:}
                       $\langle$if $\neg active_{\psi} \vee turn = \phi \rightarrow$ skip fi$\rangle$;
                       {6:}
                       critical section;
                       {7:}
                       $active_{\phi}$ := false
                 od
  //
    $\psi$: {8:} do true $\rightarrow$ {9:}
                       non critical section;
                       {10:}
                       $active_{\psi}$ := true;
                       {11:}
                       turn := $\phi$;
                       {12:}
                       $\langle$if $\neg active_{\phi} \vee turn = \psi \rightarrow$ skip fi$\rangle$;
                       {13:}
                       critical section;
                       {14:}
                       $active_{\psi}$ := false
                 od
  coend
```

Figure 7.1. Peterson's Protocol

7.1. Mutual Exclusion

A property outline for process ϕ of *crits* and property recognizer m_{mutex} (see Figure 2.4) appears in in Figure 7.2; the property outline for ψ is symmetric. The only non-trivial part of showing that Figure 7.2 is a valid property outline is showing Interference Freedom—in particular, showing that execution of ψ cannot invalidate the property assertion at control point

```
$\phi$: {1: $q_0 \sim$ *true*}
    **do** *true* → {2: $q_0 \sim$ *true*}
                non critical section;
                {3: $q_0 \sim$ *true*}
                $active_\phi$ := *true*;
                {4: $q_0 \sim active_\phi$}
                *turn* := $\psi$;
                {5: $q_0 \sim active_\phi$}
                ⟨**if** $\neg active_\psi \vee turn = \phi$ → **skip fi** ⟩;
                {6: $q_0 \sim active_\phi \wedge (turn = \phi \vee \neg active_\psi \vee pc_\psi = 11)$}
                critical section;
                {7: $q_0 \sim$ *true*}
                $active_\phi$ := *false*
    **od**
```

Figure 7.2. Mutual Exclusion Property Outline

6, since this is the only property assertion in ϕ that mentions variables altered by execution of ψ. Execution of $active_\psi$:= *true* by ψ (at control point 10) makes $pc_\psi = 11$ true, and execution of *turn* := ϕ by ψ (at control point 11) makes $turn = \phi$ true. Thus, the property assertion is not interfered with.

To prove that π satisfies the property accepted by m_{mutex}, we must first define Cs_ϕ and Cs_ψ in terms of the program state:

$$Cs_\phi \equiv 6 \le pc_\phi \le 7$$

$$Cs_\psi \equiv 13 \le pc_\psi \le 14$$

Next, we must prove Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10). We can use (6.17) to extract from the property outline a correspondence invariant for automaton state q_0:

$$\begin{aligned} C_0 \;\equiv\; & (pc_\phi{=}4 \Rightarrow active_\phi) \wedge (pc_\phi{=}5 \Rightarrow active_\phi) \wedge \\ & (pc_\phi{=}6 \Rightarrow (active_\phi \wedge (turn{=}\phi \vee \neg active_\psi \vee pc_\psi{=}11))) \wedge \\ & (pc_\psi{=}11 \Rightarrow active_\psi) \wedge (pc_\psi{=}12 \Rightarrow active_\psi) \wedge \\ & (pc_\psi{=}13 \Rightarrow (active_\psi \wedge (turn{=}\psi \vee \neg active_\phi \vee pc_\phi{=}4))) \end{aligned}$$

Simulation Basis requires that we prove

$$Init_{crits} \Rightarrow (\neg (Cs_\phi \wedge Cs_\psi) \wedge C_0) \qquad (7.1)$$

Substituting and simplifying, we find that (7.1) is valid. Simulation Induction (4.7) follows because the property outline of Figure 7.2 is a valid. Finite Acceptance, Knot Exit, and Knot Variance are vacuously satisfied because the single automaton state of m_{mutex} is both a finite-accepting and infinite-accepting state.

7.2. Starvation Freedom

In Peterson's mutual exclusion protocol, process ϕ makes a request to enter its critical section by reaching control point 5; its request is serviced when it reaches control point 6. Thus, to use property recognizer m_{starv} (Figure 2.5) to show Starvation Freedom for ϕ, we choose transition predicates:

$$Request_\phi \;\equiv\; pc_\phi{=}5$$

$$Served_\phi \;\equiv\; pc_\phi{=}6$$

A valid property outline for the protocol and m_{starv} is given in Figure 7.3. Proving Sequential Correctness and Interference Freedom is simple and is omitted here.

We extract correspondence invariants from the property outline using (6.17):

$$C_0 \;\equiv\; (pc_\phi{\neq}5 \Rightarrow (turn{=}\phi \vee turn{=}\psi)) \wedge (pc_\phi{=}4 \Rightarrow active_\phi) \wedge (pc_\phi{=}5 \Rightarrow false)$$

$$\begin{aligned} C_1 \;\equiv\; & (pc_\phi{\neq}5 \Rightarrow false) \wedge (pc_\phi{=}5 \Rightarrow (active_\phi \wedge (turn{=}\phi \vee turn{=}\psi))) \wedge \\ & (pc_\psi{\neq}12 \Rightarrow (pc_\phi{=}5 \wedge turn{=}\psi)) \wedge (pc_\psi{=}12 \Rightarrow pc_\phi{=}5) \end{aligned}$$

To prove Simulation Basis (4.6) we show that

$$Init_{crits} \Rightarrow (\neg Request_\phi \wedge C_0) \vee (Request_\phi \wedge C_1)$$

is valid. This simplifies to

$$\begin{aligned} & pc_\phi{=}1 \wedge pc_\psi{=}8 \wedge \neg active_\phi \wedge \neg active_\psi \wedge (turn{=}\phi \vee turn{=}\psi) \\ & \quad \Rightarrow (pc_\phi{\neq}5 \wedge C_0) \vee (pc_\phi{=}5 \wedge C_1) \end{aligned}$$

which is valid.

Next, we prove Finite Acceptance (4.8). There is only one non-finite-accepting state in m_{starv}, q_1. Thus, Finite Acceptance (4.8) requires that we show that

$$C_1 \Rightarrow \neg Blocked_{crits}$$

```
crits: cobegin
    $\phi$: {1: $q_0 \sim (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
        do true $\rightarrow$ {2: $q_0 \sim (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
                    non critical section;
                    {3: $q_0 \sim (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
                    $active_\phi := true$;
                    {4: $q_0 \sim active_\phi \wedge (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
                    $turn := \psi$;
                    {5: $q_0 \sim false \mid q_1 \sim active_\phi \wedge (turn=\phi \vee turn=\psi)$}
                    $\langle$if $\neg active_\psi \vee turn=\phi \rightarrow$ skip fi $\rangle$;
                    {6: $q_0 \sim (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
                    critical section;
                    {7: $q_0 \sim (turn=\phi \vee turn=\psi) \mid q_1 \sim false$}
                    $active_\phi := false$
        od
  //
    $\psi$: {8: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
        do true $\rightarrow$ {9: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
                    non critical section;
                    {10: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
                    $active_\psi := true$;
                    {11: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
                    $turn := \phi$;
                    {12: $q_0 \sim true \mid q_1 \sim pc_\phi=5$}
                    $\langle$if $\neg active_\phi \vee turn=\psi \rightarrow$ skip fi $\rangle$;
                    {13: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
                    critical section;
                    {14: $q_0 \sim true \mid q_1 \sim pc_\phi=5 \wedge turn=\psi$}
                    $active_\psi := false$
        od
  coend
```

Figure 7.3. Starvation Freedom Property Outline

is valid. It is.

There is one reject knot $\kappa = \{q_1\}$ in m_{starv}. Choose the following as a variant function v_κ for the knot.

$$v_\kappa(q) \;=\; \begin{cases} 0, & \text{if } pc_\phi \neq 5 \\ 1, & \text{if } pc_\phi = 5 \wedge pc_\psi = 12 \wedge \mathit{turn} = \phi \\ 2+((11-pc_\psi)\ \mathit{mod}\ 6), & \text{if } pc_\phi = 5 \wedge \mathit{turn} = \psi \end{cases}$$

To satisfy Knot Exit (4.9), we must prove

$$(v_\kappa(q_1)=0) \Rightarrow (\mathit{Blocked}_{crit} \vee \neg C_1).$$

This follows because $v_\kappa(q_1)=0 \Rightarrow pc_\phi \neq 5$ and $pc_\phi \neq 5 \Rightarrow \neg C_1$.

To satisfy Knot Variance (4.10), we must show that for every atomic action α:

$$\{C_1 \wedge 0 < v_\kappa(q_1) = V\} \;\; \alpha \;\; \{(\neg \mathit{Served}_\phi \wedge C_1) \Rightarrow v_\kappa(q_1) < V\} \qquad (7.2)$$

Since $v_\kappa(q_1)=1 \Rightarrow (pc_\phi = 5 \wedge pc_\psi = 12 \wedge \mathit{turn} = \phi)$, and $(pc_\phi = 5 \wedge C_1) \Rightarrow \mathit{active}_\phi$, it suffices to prove

$$\{\mathit{active}_\phi \wedge pc_\phi = 5 \wedge pc_\psi = 12 \wedge \mathit{turn} = \phi\} \;\; \alpha \;\; \{(\neg \mathit{Served}_\phi \wedge C_1) \Rightarrow v_\kappa(q_1) < 1\} \qquad (7.3)$$

for each atomic action α. Only the atomic actions at control points 5 and 12 are potentially enabled in the precondition of (7.3), and from $\mathit{active}_\phi \wedge \mathit{turn} = \phi$, we conclude that the one at 12 is not enabled. Since $\neg$*Served* is *false* after the atomic action at control point 5 is executed, the postcondition of (7.3) is *true* and the triple is valid.

Next, we show that (7.2) is valid if $v_\kappa(q_1)=2$. From $v_\kappa(q_1)=2$, we infer $pc_\phi = 5 \wedge \mathit{turn} = \psi$ and since $2+((11-11)\ \mathit{mod}\ 6 = 2$, $pc_\psi = 11$. Thus, if suffices to show that

$$\{pc_\phi = 5 \wedge \mathit{turn} = \psi \wedge pc_\psi = 11\} \;\; \alpha \;\; \{(\neg \mathit{Served}_\phi \wedge C_1) \Rightarrow v_\kappa(q_1) < 2\} \qquad (7.4)$$

is valid. Only the atomic action at control points 5 and 11 are enabled in the precondition of (7.4), so they are the only ones for which (7.4) is not trivially valid. Executing the atomic action at control point 5 makes $pc_\phi = 6$, hence the postcondition of (7.4) is *true* and the triple valid; executing the atomic action at control point 11 makes $pc_\psi = 12 \wedge \mathit{turn} = \phi$, which decreases $v_\kappa(q_1)$ to 1.

Finally, we show that (7.2) is valid if $v_\kappa(q_1) > 2$. If $v_\kappa(q_1) > 2$ then the atomic action at control point 5, as well as an action at 9, 10, or 12−14 must be enabled. As already argued, executing the atomic action at 5 decreases $v_\kappa(q_1)$ to 0. Executing an atomic action at 9, 10, 12, 14 also decreases $v_\kappa(q_1)$, since by reaching the next control point, the value of $2+((11-pc_\psi)\ \mathit{mod}\ 6)$ is decreased. Execution starting from 13 causes the value of $2+((11-pc_\psi)\ \mathit{mod}\ 6)$ to be decreased provided control point 14 is reached. Thus, our proof of Starvation Freedom is correct only if ψ is guaranteed to exit its critical section after entering it.

7.3. First-come First-served

A property recognizer for First-come First-served for *crits* is given in Figure 7.4. Transition predicates $Request_\phi$ and $Served_\phi$ are as defined above for Starvation Freedom; the remaining two transition predicates used in m_{fcfs} are:

$Request_\psi \;\equiv\; pc_\psi = 12$

$Served_\psi \;\equiv\; pc_\psi = 13$

A property outline for *crits* and m_{fcfs} appears in Figure 7.5. Showing that the Property Outline is valid is straightforward; we do not give the details here. Informally, the correspondence invariants characterize states as follows.

C_0: either ϕ does not have a pending request or ψ has a prior request pending.

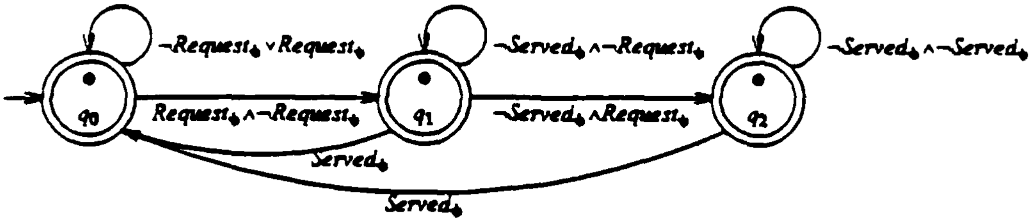
C_1: ϕ has a pending request and ψ does not.

C_2: both ϕ and ψ have pending requests and the one from ϕ was prior to the one from ψ.

Simulation Basis (4.6) follows trivially. The remaining obligations—Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10)—are vacuously true because every automaton state in m_{fcfs} is both finite-accepting and infinite-accepting. Thus, the proof is completed.

8. Non-deterministic Property Recognizers

The proof obligations of section 4 concern properties specified by deterministic property recognizers. We now address the problem of proving that every history of a program π is accepted by some given non-deterministic property recognizer m_{ND}. Two approaches are discussed. In the first, proof obligations are extracted directly from m_{ND}. In the second, a deterministic property recognizer m_D is constructed that accepts every history of π accepted by m_{ND}, but not necessarily every sequence of states accepted by m_{ND}. Then, proof obligations are extracted from m_D. The relative completeness result of the Appendix establishes



$\neg Request_\phi \vee Request_\psi$
$\neg Served_\phi \wedge \neg Request_\psi$
$\neg Served_\phi \wedge \neg Served_\psi$
q_0
$Request_\phi \wedge \neg Request_\psi$
q_1
$\neg Served_\phi \wedge Request_\psi$
q_2
$Served_\phi$
$Served_\phi$

Figure 7.4. m_{fcfs}

```
crits: cobegin
    $\phi$: {1: $q_0 \sim true \mid q_1 \sim false \mid q_2 \sim false$}
        do  true  → {2: $q_0 \sim true \mid q_1 \sim false \mid q_2 \sim false$}
                    non critical section;
                    {3: $q_0 \sim true \mid q_1 \sim false \mid q_2 \sim false$}
                    $active_\phi$ := true;
                    {4: $q_0 \sim active_\phi \mid q_1 \sim false \mid q_2 \sim false$}
                    turn := $\psi$;
                    {5: $q_0 \sim Request_\psi \wedge turn = \psi \wedge active_\phi \mid q_1 \sim \neg Request_\psi \wedge active_\phi \mid$
                        $q_2 \sim Request_\psi \wedge turn = \phi \wedge active_\phi$}
                    ⟨if $\neg active_\psi \vee turn = \phi$ → skip fi ⟩;
                    {6: $q_0 \sim true \mid q_1 \sim false \mid q_2 \sim false$}
                    critical section;
                    {7: $q_0 \sim true \mid q_1 \sim false \mid q_2 \sim false$}
                    $active_\phi$ := false
        od
  //
    $\psi$: {8: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
        do  true  → {9: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
                    non critical section;
                    {10: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
                    $active_\psi$ := true;
                    {11: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
                    turn := $\phi$;
                    {12: $q_0 \sim Request_\phi \Rightarrow (turn = \psi \wedge active_\phi) \mid$
                         $q_1 \sim false \mid q_2 \sim active_\phi \wedge turn = \phi$}
                    ⟨if $\neg active_\phi \vee turn = \psi$ → skip fi ⟩;
                    {13: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
                    critical section;
                    {14: $q_0 \sim Request_\phi \Rightarrow active_\phi \mid q_1 \sim Request_\phi \Rightarrow active_\phi \mid q_2 \sim false$}
                    $active_\psi$ := false
        od
  coend
```

Figure 7.5. First-come First-served Property Outline

that the second appr:acn always works, provided the program has a finite state space; however, the first approacii is often simpler and more convenient.

8.1. Extracting Proo/ Obligations

The proof obligations of section 4 are based on two assumptions that hold for deterministic property recognizers:

(1) There is a single start state.

(2) Disjoint transition predicates label arcs emanating from each automaton state.

These assumptions need not hold for non-deterministic property recognizers. However, given a non-deterministic property recognizer that does not satisfy assumption (1), it is easy to construct one that does. Thus, in adapting the proof obligations developed in section 4 for use with properties specified by non-deterministic property recognizers, we need only be concerned with assumption (2).

Assumption (2) is used in section 4 to combine the constraints on correspondence invariants with the proof obligations that prevent undefined transitions. In particular, (4.1) is merged with (4.3) to form Simulation Basis (4.6), and (4.2) is merged with (4.4) to form Simulation Induction (4.7). Since this merging is not possible *when transition predicates* are not disjoint, the reasoning of section 4 dictates that for a given program π and non-deterministic property recognizer m_{ND}, showing (4.1), (4.2), (4.3), (4.4), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10), ensures that every history of π is accepted by m_{ND}.

Unfortunately, these proof obligations may be too strong—not all programs that satisfy m_{ND} will satisfy (4.1), (4.2), (4.3), (4.4), (4.8), (4.9), and (4.10) because these obligations ensure that for any history of the program, *every* run of m_{ND} is accepting. Recall, a property recognizer accepts an infinite sequence provided a single run is accepting. With a deterministic property recognizer, each input results in only a single run, so ensuring that every run is accepting is equivalent to ensuring that the single run is. With a non-deterministic property recognizer, there may be multiple runs. Thus, *for non-deterministic property recognizers*, the proof obligations are more restrictive than necessary.

8.2. Refining Non-deterministic Recognizers

Non-deterministic property recognizers can specify properties that cannot be specified by deterministic ones [Eilenberg 74]. However, each program π (with a finite state space) that satisfies a property P_{ND}, accepted by a non-deterministic property recognizer m_{ND}, must also satisfy a property P_D, where $P_D \subseteq P_{ND}$ and P_D is specified by a deterministic property recognizer m_D.[4] Thus, to prove that π satisfies a property ND specified by m_{ND}, it suffices to construct m_D and prove that π satisfies it. We call m_D a *deterministic refinement* of m_{ND}.

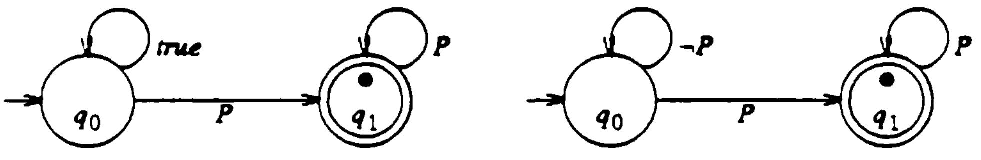
The construction of m_D involves repeatedly modifying m_{ND}, using the techniques described below, so that it becomes progressively more deterministic. Clearly, valid modifications must never cause the resulting property recognizer to accept sequences not accepted by the original one; they can, however, cause fewer sequences to be accepted. Satisfying the proof obligations for the deterministic refinement ensures that all histories of the program are accepted by the original property recognizer m_{ND}.

Modifications for obtaining a deterministic refinement fall into two classes: those that result in an automaton that accepts the same sequences as the original; and those that result in an automaton that accepts fewer sequences than the original. The second class of modifications is needed because some non-deterministic property recognizers do not have deterministic equivalents.

By removing transitions from m_{ND}, the resulting property recognizer is more deterministic and can accept no sequence that would not have been accepted by m_{ND}. Thus, this form of modification is one way towards constructing a deterministic refinement.

Pruning: Delete transitions in the property recognizer.

Frequently, Pruning is performed by strengthening transition predicates based on knowledge of the program state. This form of Pruning is illustrated in Figure 8.1.



Before

After

Figure 8.1. Pruning

Here, transitions from q_0 to itself under program states that satisfy P have been pruned.

A second modification that makes a property recognizer more deterministic is to combine automaton states.

Combining: Combine states if it does not permit additional sequences to be accepted.

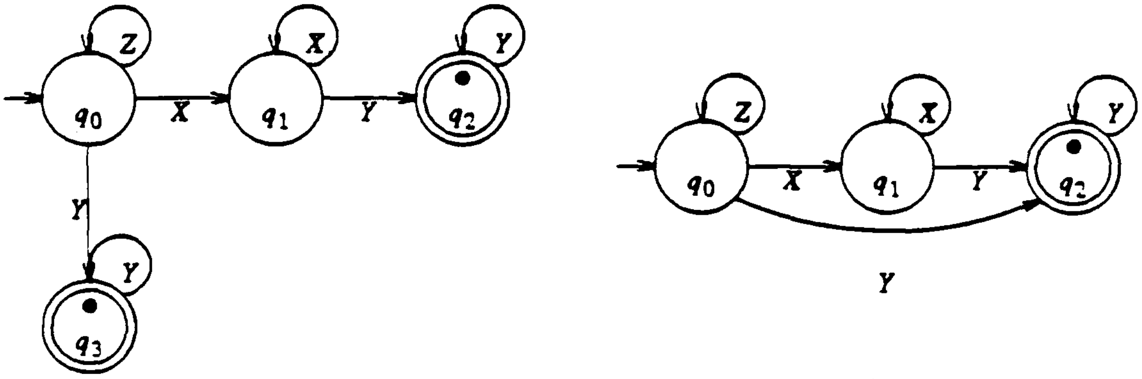
When combining two states q' and q'', all transitions into q' and q'' terminate at a new state

[4] The proof of this appears in the Appendix as part of the completeness result.

q. If a non-deterministic choice selected between q and q' in the original property recognizer, then that choice is no longer non-deterministic in the resulting one. Two states q' and q'' can be combined provided:

> **Combining Congruent States.** If two states q' and q'' are congruent then they can be combined and the resultant property recognizer will accept the same set of sequences. Two states q' and q'' are *congruent* if and only if
>
> C1: neither or both are finite-accepting,
>
> C2: neither or both are infinite-accepting,
>
> C3: if there is a transition from q' to q under program state s then there also is a transition from q'' to some state congruent to q under program state s.

An example of this is illustrated in Figure 8.2. There, q_2 and q_3 are combined.



After

Before

Figure 8.2. Combining

When C1 or C2 of Combining Congruent States does not hold, it is sometimes possible to promote a non-accepting state to being an accepting state without changing the set of sequences accepted by the property recognizer.

> **Finite-accepting Promotion.** A non-finite-accepting state q can be promoted to being finite-accepting if for every run that ends in q there is another run on the same input that ends in a finite-accepting state.

> **Infinite-accepting Promotion.** A non-infinite-accepting state q can be promoted to being infinite-accepting if for every run that contains q infinitely-often there is a run (perhaps the same one) on the same input that contains some infinite-accepting state infinitely

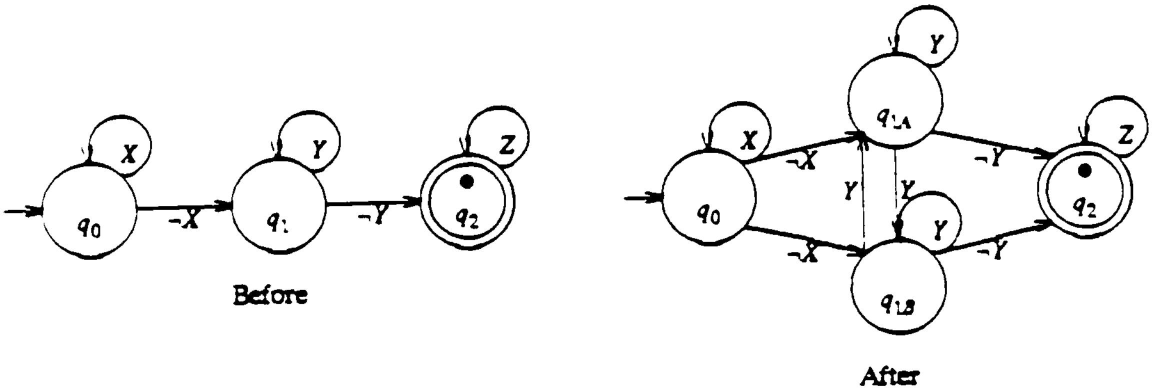
often.

Finally, an automaton state may serve many roles. By splitting such a state into several copies, we can separate these roles and then use Pruning to remove transitions or Combining to combine some of the copies with other automaton states.

Splitting: Replicate an automaton state and all transitions into and out of it.

Splitting does not change the set of sequences accepted by a property recognizer, but it does put the recognizer into a form where Pruning and/or Combining can be used to move towards a deterministic refinement. Splitting is illustrated in Figure 8.3.

It is not always necessary to construct the actual deterministic refinement of a given non-deterministic property recognizer. Rather, it suffices to use Pruning, Combining, and Splitting to obtain a non-deterministic property recognizer for a property that is also accepted by some deterministic property recognizer. We can then apply one of the known (automatic) procedures to produce[5] a deterministic property recognizer that is equivalent to the given non-deterministic one [Landweber 69].



Before

After

Figure 8.3. Splitting

[5] Such procedures also indicate if there is no deterministic property recognizer for the given non-deterministic one. Then additional Pruning, Combing, and Splitting must be done.

9. Discussion

We have shown how to decompose a property into proof obligations. Since properties and proof obligations can be formalized using temporal logic, our approach describes how to break up the task of showing that a program satisfies one temporal formula—the property—into showing that the program satisfies a number of simpler temporal formulas—the proof obligations. Simulation Basis (4.6), Finite Acceptance (4.8), and Knot Exit (4.9) are temporal formulas because they are predicate logic formulas. The remaining two proof obligations, Simulation Induction (4.7) and Knot Variance (4.10), can be formulated in temporal logic, as

Temporal Simulation Induction: For all i: $q_i \in Q$:

$$\Box(C_i \Rightarrow \bigcirc(\bigvee_{j:q_j \in Q}(T_{ij} \wedge C_j))), \tag{9.1}$$

Temporal Knot Variance: For all reject knots κ and all $q_i \in \kappa$:

$$\Box((C_i \wedge 0 < v_\kappa(q_i) = V) \Rightarrow \bigcirc(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V))) \tag{9.2}$$

where $\Box$ denotes the temporal operator "henceforth" and $\bigcirc$ denotes "next".

Other investigations into decomposing temporal properties include [Barringer et al. 84], [Gerth 84], [Jones 83], [Misra et al. 82], [Nguyen et al. 85] and [Stark 84]. Most of that work is concerned with decomposing various classes of global temporal properties of a system into local properties of the system components, resulting in so-called compositional proof systems. The work in [Gerth 84] is most similar to ours in that the primitive formulas into which temporal properties are decomposed resemble triples. That work, however, is concerned only with finite sequences (both as properties and programs) and therefore does not address the problem we are most concerned with.

We chose to express the proof obligations as triples rather than as temporal logic formulas because our experience is that people have less trouble understanding and manipulating triples. Moreover, the relation between triples and the program text is always clear—when a proof obligation formulated as a triple cannot be proved, there is little question where in the program to start looking. This is not the case for formulas of temporal logic, because they do not explicitly mention the program. Finally, we hope to integrate our approach with methods to develop a program and its proof of correctness hand-in-hand, as discussed in [Dijkstra 76] [Gries 81]. These methods are formulated in terms of triples, so it made sense for us to remain in that framework.

Considering our proof obligations from a temporal viewpoint does offer some insights. Temporal Knot Variance (9.2) requires that execution of *every* atomic action cause the value of a variant function to decrease, thereby ensuring progress is made towards accepting the history. Without making assumptions about fairness, this is the only way to ensure that all infinite histories leave a reject knot because an atomic action that does not decrease any variant function can be repeated indefinitely, resulting in a history that is not accepted by the property recognizer. Thus, while we would be happy to establish

$$\Box((C_i \wedge 0 < v_\kappa(q_i) = V) \Rightarrow \Diamond(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V))), \tag{9.3}$$

(where $\Diamond$ denotes eventually), without making fairness assumptions, we are forced to demonstrate

$$\Box((C_i \wedge 0 < v_\kappa(q_i) = V) \Rightarrow \bigcirc(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V))). \tag{9.4}$$

However, if we can make assumptions about fairness, then we need not prove (9.4), in order to establish (9.3). Instead, it suffices to prove that certain *helpful processes* that do decrease the variant function are eventually executed and that executing other processes does not increase the variant function. This method is formalized as temporal logic inference rules in [Manna & Pnueli 84]—one rule for each type of fairness (e.g. weak fairness, strong fairness)—and can be adapted to our approach by replacing Knot Variance (4.10) with the hypotheses of the appropriate inference rule. These hypothesis are easily formulated as predicate logic formulas and triples. This, then, provides a second way in our approach to prove a property P under a fairness assumption F. The first (section 4), was to construct the property recognizer for $F \Rightarrow P$ and show that the proof obligations it defines are satisfied; the second, is to construct a property recognizer for P and extract proof obligations from it, except with the Knot Variance (4.10) obligation replaced by the hypotheses from the appropriate temporal logic inference rule.

One difference between our approach and most temporal logic verification methods is the treatment of terminating executions. We handle terminating executions by explicitly dealing with finite sequences of program states; it is inconvenient to deal with finite sequences using temporal logics that include a "next" operator, so finite sequences are usually extended to be infinite sequences. Unfortunately, this extension can cause problems because the infinite sequence might not satisfy a property that the original (finite) one did. For example, a common way to extend a finite sequence to an infinite one is by replicating the last state. A property like "the value of the program counter changes between two successive states", though *true* of a finite sequence, does not hold for an infinite sequence obtained by replicating the last state of a finite sequence. Other ways to extend finite sequences have similar problems.

Another, related, approach to verifying that a program satisfies a property is *model checking* [Clarke et. al. 83] [Emerson & Lei 85] [Lichtenstein & Pnueli 85], where a program π is viewed as specifying a *Kripke structure* K_π. K_π is a model for P if and only if π satisfies P. Thus, to determine if π satisfies P it suffices to check whether K_π is a model for P, and this amounts to checking each state in the state space to see which sub-formulas of P hold in that state. Determining whether if K_π is a model for P requires time linear in both the length of P and the size of the program state space.

Recently, [Vardi & Wolper 85] observed that K_π can be viewed as a Buchi automata[6] that accepts exactly the histories of π. From this automaton and one that recognizes sequences satisfying $\neg P$, a Buchi automaton $m_{\pi \wedge \neg P}$ can be constructed that accepts all histories of π not satisfying P. The decision procedure for the emptyness problem for $m_{\pi \wedge \neg P}$ can then be used to determine if π satisfies P; the decision procedure is exponential in the length of P and linear in the size of the program state space.

The drawback to both these methods is that they require time linear in the size of the state space. (The fact that the second method is exponential in the length of P is inconsequential due to the relative size of the program state space.) They are practical only for those applications where the program state space is of a manageable size. In our approach, rather than check every state in the state space, the state space is partitioned into equivalence classes defined by the correspondence invariants. The number of correspondence invariants is exponential in the length of P, since there is one for each state in m_P; the number of proof obligations is linear in the size of the program. Thus, with our method, the number of proof obligations incurred for a deterministic property is exponential in the length of P and linear in the size of the program. Since the size of the program is likely to be substantially smaller than the size of the state space, our approach is rather attractive.[7] Even for non-deterministic properties, the number of proof obligations incurred with our approach is bounded by the size of the state space (see Appendix). Thus, our approach is comparable to the model checking approaches for this case.

Of course, verification is only necessary if synthesis is not possible. Techniques to synthesize the synchronization portion of a finite-state concurrent program from a propositional temporal logic specification are given in [Clarke & Emerson 81] and [Manna & Wolper 84]. The latter technique is most closely related to the work of this paper, since it is based on linear time temporal logic. In it, a ***model graph*** for a property P is constructed and then converted into a program. This model graph is just a property recognizer. Restriction to propositional specifications is not a problem for synchronizers, but is not sufficient for specifying many properties of programs; e.g. the relation between the program's input and output.

[6] Recall, Buchi automata are special cases of property recognizers.

[7] We assume that the cost of deciding the validity of a Hoare triple is constant. This is reasonable for purposes of comparison because in the model checking approach the ability to decide the validity of an implication in constant time follows from the restriction to propositional temporal logic.

10. Conclusions

A new approach to proving temporal properties of concurrent programs was described. The approach is based on specifying properties using automata, called property recognizers. Property recognizers are quite expressive—any linear-time temporal logic formula can be formulated as a property recognizer. Proof obligation for a property are extracted directly from the recognizer for that property. The proof obligations are predicate logic formulas and triples. Thus, temporal inference is not necessary for proving temporal properties. In fact, the same techniques that work for proving total correctness of sequential programs [Hoare 69] [Dijkstra 76] can be used for proving arbitrary temporal properties of concurrent ones. When proving total correctness of a loop in a sequential program, a loop invariant and variant function must be devised and checked. When our method is used to prove that some arbitrary temporal property holds for a concurrent program, correspondence invariants and variant functions must be devised and checked.

Our approach was illustrated on some standard examples: incrementing x by 2 in parallel [Owicki & Gries 76] and Mutual Exclusion, Starvation Freedom, and First-come, First-served for Peterson's solution to the critical section problem [Peterson 81]. Property outlines were proposed as a succinct way to represent a program and its correspondence invariants for a given property recognizer.

Acknowledgments

D. Gries, L. Lamport, and P. Panangaden made helpful comments on an earlier draft of this paper.

References

[Alpern 86] Alpern, B. Constructing proof obligations. Ph.D. Thesis. Department of Computer Science, Cornell University. In preparation.

[Barringer et al. 84] Barringer, H, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proc. Sixteenth Annual Symposium on Theory of Computing*, Washington, D.C., April 1984, 51-63.

[Carke & Emerson 81] Carke, E.M. and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logic of Programs* (D. Kozen ed.), Lecture Notes in Computer Science Vol. 131, Springer Verlag, Berlin, 1981, 52-71.

[Carke et. al. 83] Carke, E.M., E.A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1983, 117-126.

[Dijkstra 76] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[Eilenberg 74] Eilenberg, S. *Automata, Languages and Machines, Vol A.* Academic Press, New York, 1974.

[Emerson & Lei 85] Emerson, E.A., and C-L. Lei. Modalities for model checking: Branching time strikes back.

Proceedings of the 12th ACM Symposium on Principles of Programming Languages, New Orleans, January 1985, pp. 84-96.

[Gerth 84] Gerth, R. Transition logic. *Proc. Sixteenth Annual Symposium on Theory of Computing*, Washington, D.C., April 1984, 39-50.

[Gries 81] Gries, D. *The Science of Programming*. Springer-Verlag, NY, 1981.

[Hoare 69] Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969) 576-580.

[Hopcroft & Ullman 79] Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[Jones 83] Jones, C.B. Specification and design of (parallel) programs. *Information Processing '83*, (R.E.A. Mason, ed.) North-Holland Publishing Company, Amsterdam, 1983, 321-332.

[Lamport 80] Lamport, L. The 'Hoare logic' of concurrent programs. *Acta Informatica 14*, 1 (1980) pp. 21-37.

[Lamport 83a] Lamport, L. What good is temporal logic. *Information Processing '83*, (R.E.A. Mason, ed.) North-Holland Publishing Company, Amsterdam, 1983, 657-668.

[Lamport 83b] Lamport, L. Specifying concurrent program modules. *ACM TOPLAS 6*, 2 (April 1983), 190-222.

[Lamport & Schneider 84] Lamport, L. and F.B. Schneider. The 'Hoare Logic' of CSP, and All That. *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 281-296.

[Landweber 69] Landweber, L.H. Decision problems for ω-automata. *Math. System Theory 3*, (1969), 376-384.

[Lichtenstein & Pnueli 85] Lichtenstein, O. and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985, 97-107.

[Manna & Pnueli 81a] Manna, Z. and A. Pnueli. Verification of concurrent programs: The temporal framework. *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1981, 141-154.

[Manna & Pnueli 81b] Manna, Z. and A. Pnueli. Verification of concurrent programs: Temporal proof principles. *Logic of Programs* (D. Kozen ed.), Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, Berlin, 1981, 200-252.

[Manna & Pnueli 83] Manna, Z., and A. Pnueli. How to cook a temporal proof system for your pet language. *Proc. of the Symposium on Principles of Programming Languages*, ACM, Austin, Jan. 1983.

[Manna & Pnueli 84] Manna, Z. and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming 4* (1984), 257-289.

[Manna & Wolper 84] Manna, Z and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems 6*, 1 (Jan. 1984), 68-93.

[Misra et al. 82] Misra, J., K.M. Chandy, and T. Smith. Proving safety and liveness of communicating processes with examples. *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982, 157-164.

[Nguyen et al. 85] Nguyen, V., D. Gries, S. Owicki. A model and temporal proof system for networks of processes. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, Jan. 1985, 121-131.

[Owicki & Gries 76] Owicki, S.S. and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica 6*, (1976), 319-340.

[Owicki & Lamport 82] Owicki, S.S. and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 455-496.

[Peterson 81] Peterson, G.L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (June 1981), 115-116.

[Pnueli 77] Pnueli, A. The temporal logic of programs. *Proc of the 18th Symposium on the Foundations of Computer Science*, IEEE, Providence R.I., Nov. 1977, 46-57.

[Pnueli 86] Pnueli, A. In transition from global to modular temporal reasoning about programs. In *Current Trends in Concurrency*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1896, To appear.

[Pnueli & Manna 83] Pnueli, A. and Z. Manna. Proving precedence properties: The temporal way. *Proc 10th*

Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science Vol. 154, Springer Verlag, Berlin, 1983, 490-510.

[Stark 84] Stark, E.W. Foundations of a theory of specification for distributed systems. Ph.D. Thesis, M.I.T. Laboratory for Computer Science. MIT/LCS/TR-342, August 1984.

[Vardi & Wolper 85] Vardi, M.Y. and P. Wolper. Applications of temporal logic: An automata-theoretic perspective. In preparation.

[Wolper 83] Wolper, P. Temporal logic can be more expressive. *Information and Control 56*, 1-2 (1983), 72-99.

[Wolper 84] Wolper, P. The tableau method for temporal logic: An overview. Unpublished manuscript.

Appendix: Soundness and Relative Completeness

The soundness and relative completeness of our approach is shown below. We first show that the proof obligations of section 4 for deterministic property recognizers are sound. We then show that they are complete relative to predicate logic and Hoare's partial correctness logic. Since partial correctness logic is known to be complete relative to predicate logic, our proof obligations are complete relative to predicate logic. Next, we show that the proof obligations of section 8 for non-deterministic property recognizers are also sound, and finally that they are complete relative to our approach for deterministic properties.

Deterministic Property Recognizers

Soundness Theorem: If for a program π and deterministic property recognizer m_P for property P there are correspondence invariants and variant functions such that Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) are valid, then π satisfies P.

Proof. Assume that the proof obligations are valid for some correspondence invariants and variant functions and that σ is a history of π. We must show that σ satisfies P.

By induction on n,

$$\delta^*(q_0, \sigma[..n]) = q_i \Rightarrow C_i(\sigma[n])$$

due to Simulation Basis (4.6) and Simulation Induction (4.7). A similar inductive argument shows that m_P cannot attempt an undefined transition when reading $\sigma[n]$.

We now show that if σ is finite then it is accepted by m_P. Without loss of generality, let $\sigma[n]$ be the final state of σ. We must show $\delta^*(q_0, \sigma[..n]) \in Q_{fin}$. Due to Finite Acceptance (4.8), if $\delta^*(q_0, \sigma[..n])$ is a non-finite-accepting state, then π cannot be blocked in $\sigma[n]$ and this contradicts the assumption that $\sigma[n]$ is the final state of σ. Thus, we conclude that $\delta^*(q_0, \sigma[..n])$ is a finite-accepting state, and, by definition, m_P accepts σ, hence σ satisfies P.

Finally, we show that if σ is infinite then it is accepted by m_P. By Knot Exit (4.9) and Knot Variance (4.10), if m_P enters a reject knot κ upon reading $\sigma[n]$, then it must exit κ before reading the $n+\nu_\kappa(\delta^*(q_0, \sigma[..n]), \sigma[n])^{th}$ symbol of σ. By the definition of a reject knot, m_P cannot reenter κ after exiting it without first entering an infinite-accepting state.

Since there are finitely many reject knots and σ is infinite, m_P must enter an infinite-accepting state infinitely often. Thus, by definition, m_P will accept σ, hence σ satisfies P. □

Relative Completeness Theorem: If a program π satisfies a property P that is accepted by a deterministic property recognizer m_P, then there exist correspondence invariants and variant functions, for which Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) are valid.

Proof. Assume m_P accepts every history of π. We must show that (4.6)−(4.10) for π and m_P are valid.

Chose correspondence invariants and variant functions as follows. Let H_π be the set of histories of π. First, for each automaton state q_l, define

$$C_i(s) \;\; = \;\; (\exists \sigma,\, n\text{:}\;\; \sigma \in H_\pi,\, 0 \le n\text{:}\;\; s = \sigma[n] \,\wedge\, \delta^*(q_0,\, \sigma[..n]) = q_l).$$

Thus, $C_i(s)$ holds for a program state s if and only if there is some history of π in which s caused m_P to make a transition to q_l. Next, for each reject knot κ and each $q_l \in \kappa$, define

$$v_\kappa(q_i,\, s) \;\; = \;\; \begin{cases} 0, & \text{if } \mathit{Blocked}_\pi(s) \vee \neg C_i(s) \\ 1 + \max\limits_{v}(\exists \sigma,\, n\text{:}\;\; \sigma \in H,\, 0 \le n\text{:}\;\; s = \sigma[n] \,\wedge\, \delta^*(q_0,\, \sigma[..n]) = q_l & \\ \qquad \wedge\, \neg \mathit{Blocked}_\pi(\sigma[n+v]) \,\wedge\, (\forall j\text{:}\;\; 0 \le j \le v\text{:}\;\; \delta^*(q_0,\, \sigma[..n+j]) \in \kappa)) & \\ \qquad \text{if } \neg \mathit{Blocked}_\pi(s) \,\wedge\, C_i(s) & \end{cases}$$

Thus, $v_\kappa(q_l,\, s)$ is the maximum number of atomic actions π can execute when in state s and m_P is in q_i before m_P will halt or leave κ.

It remains is to prove that (4.6)−(4.10) are valid with these correspondence invariants and variant functions. We consider each proof obligation in turn.

Simulation Basis (4.6). Since π satisfies P, every initial state of π must satisfy some transition predicate T_{0j}. By construction, this initial state will also satisfy C_j. Thus, (4.6) is valid.

Simulation Induction (4.7). Consider any program history σ and suppose $\delta^*(q_0,\, \sigma[..n]) = q_i$ for some n. By construction, $C_i(\sigma[n])$. Consider an atomic action α from A_π that terminates in a state s' when started in state $\sigma[n]$. Clearly, $\sigma\, s'$ is the prefix of some history σ' of π. Since m_P accepts every history of π, m_P must accept σ', so there must exist an automaton state q_j such that $\sigma'[n+1]$ satisfies T_{ij}. By construction, $C_j(\sigma'[n+1])$. So, we have shown $\{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q}(T_{ij} \wedge C_j)\}$ is valid for any atomic action that terminates when started in a state satisfying C_i. Since $\{C_i\}\ \alpha\ \{T_{ij} \wedge C_j\}$ is valid for any atomic action α that does not terminate when started in a state satisfying C_i, we have shown that (4.7) is valid.

Finite Acceptance (4.3). Consider any program state $\sigma[n]$ in some history σ of π. Suppose $\delta^*(q_0, \sigma[..n]) = q_j$. Thus, by construction $C_j(\sigma[n])$. If $q_j \in Q - Q_{fin}$, then $\sigma[n]$ also satisfies $\neg Blocked_\pi$. Otherwise, $\sigma[n]$ would have to be the final state of σ, which would cause m_P to reject σ, contradicting the assumption that every history of π is accepted by m_P. Thus, $C_j \Rightarrow \neg Blocked_\pi$ is valid, so (4.8) is valid.

Knot Exit (4.9). The proof that (4.9) is valid is trivial, by construction of ν_κ.

Knot Variance (4.10). If α does not terminate when started in a state satisfying some correspondence invariant C_i for an automaton state $q_i \in \kappa$, then

$$\{C_i \wedge \nu_\kappa(q_i) = V\}\ \alpha\ \{\bigwedge_{j : q_j \in \kappa} ((T_{ij} \wedge C_j) \Rightarrow \nu_\kappa(q_j) < V)\} \tag{10.1}$$

is trivially valid.

Suppose α does terminate and terminates in state s' when started in state s. Thus, there must exist a history σ_1 and an integer n_1 such that $\sigma_1[n_1] = s$ and $\delta^*(q_0, \sigma_1[..n_1]) = q_i$. There also must exist a history σ_2 and an integer n_2 such that $\sigma_2[n_2] = s'$, $\delta^*(q_0, \sigma_2[..n_2]) = q_j$, $(\forall j\!: 0 \le j \le \nu_\kappa(q_j, s')\!: \delta^*(q_0, \sigma_2[..n_2+j] \in \kappa)$, and $\neg Blocked_\pi(\sigma_2[\nu_\kappa(q_j, s')])$. Let $\sigma = \sigma_1[0..n_1]\sigma_2[n_2..]$. Since α terminates in s' when started in state s, σ is a history of π. By the construction of ν_κ, we conclude $\nu_\kappa(q_j, s') + 1 \le \nu_\kappa(q_i, s)$. So, (10.1) is valid. □

Non-deterministic Property Recognizers

The Soundness Theorem for non-deterministic property recognizers shows that constructing a deterministic refinement suffices for proving the non-deterministic property of interest. The Soundness Theorem for deterministic property recognizers, then allows us to conclude that satisfying the proof obligations extracted from this deterministic refinement are sufficient. Completeness for non-deterministic property recognizers involves showing that if a program π satisfies a property specified by a non-deterministic property recognizer m_{ND}, then it is always possible to construct a deterministic refinement of m_{ND} by using Combining, Pruning, and Splitting.

> **Soundness Theorem:** If a non-deterministic property recognizer m_{ND} for a property ND can be refined to a deterministic property recognizer m_D for a property D by using Pruning, Splitting, or Combining, then if program π satisfies D, it will also satisfy ND.

Proof. Suppose m_D can be obtained from m_{ND} using a single refinement step. If Splitting is used, then m_D and m_{ND} accept exactly the same sequences. If Combining is used, then by the definition of Combining m_D and m_{ND} accept exactly the same sequences. Finally, if Pruning is used, then m_{ND} accepts every sequence accepted by m_D because Pruning can only result in a refinement that rejects more sequences than the original. Thus, if π satisfies property D, it must also satisfy ND. The theorem then follows by induction of the number of refinement steps needed to obtain m_D from m_{ND}. □

Relative Completeness Theorem: If program π has a finite state space and satisfies some property ND that is accepted by a non-deterministic property recognizer m_{ND}, then there exists a deterministic refinement m_D of m_{ND} that π satisfies.

Proof. First, we construct a deterministic property recognizer m_π that accepts H_π, the histories of π. Define m_π to be $\langle S_\pi, S_\pi \cup \{start\}, \{start\}, S_\pi, Blocked_\pi, \delta_\pi \rangle$, where S_π is the set of program states of π and

$\delta_\pi(start, s) = s$ iff s satisfies $Init_\pi$, and

$\delta_\pi(s, s') = s'$ iff there is an atomic action of π enabled in s that terminates in s'.

Clearly, m_π accepts exactly the histories of π.

We can use m_π to refine $m_{ND} = \langle S_\pi, Q, Q_0, Q_{inf}, Q_{fin}, \delta_{ND} \rangle$. Let $m_{ND\times\pi}$ be the property recognizer $\langle S_\pi, Q \times (S_\pi \cup \{start\}), Q_0 \times \{start\}, Q_{inf} \times S_\pi, Q_{fin} \times Blocked_\pi, \delta_{ND\times\pi} \rangle$, where

$\langle q', s' \rangle \in \delta_{ND\times\pi}(\langle q, s \rangle, s')$ iff $q' \in \delta_{ND}(q, s')$ and $\delta_\pi(s, s') = s'$.

Note that $m_{ND\times\pi}$ can be obtained by Splitting each state of m_{ND}, into one copy for each state of m_π and then using Pruning.

$m_{ND\times\pi}$ accepts exactly those sequences that are histories of π (hence, accepted by m_π) and accepted by m_{ND}. Since π satisfies ND, every history of π is accepted by m_{ND}. Thus, $m_{ND\times\pi}$ recognizes the same set of sequences as m_π. We can now use Combining to obtain m_π from $m_{ND\times\pi}$—all states of the same second component are combined together. Since m_π is deterministic and accepts every history of π, we have shown how to obtain a deterministic refinement for m_{ND}. □